UNIVERSITÉ DE SHERBROOKE
Faculté de génie
Département de génie électrique et de génie informatique

AGENTS MOBILES NATIFS POUR SYSTÈMES EMBARQUÉS

Thèse de doctorat
Spécialité: génie électrique

Mohamed Ali IBRAHIM

Jury: Abdenour BOUZOUANE, Université du Québec à Chicoutimi, Département d'Informatique et Mathématique

Gabriel GIRARD, Université de Sherbrooke, Département d'Informatique

Ruben GONZALEZ-RUBIO, Université de Sherbrooke, Département de génie électrique et de génie informatique

Philippe MABILLEAU (directeur de la thèse), Université de Sherbrooke, Département de génie électrique et de génie informatique

Sherbrooke (Québec) Canada Avril 2014

À mon épouse Oubah, à mon fils Waaberi, à mes filles Hasna et Halima (qui n'a que 2 mois)

RÉSUMÉ

L'objectif de ce projet de recherche est de développer une technologie d'agents mobiles pour systèmes embarqués. Dans un premier temps, une plateforme d'agents mobiles pour systèmes embarqués homogènes est réalisée et, ensuite dans un deuxième temps, une application d'informatique diffuse qui exploite la mobilité du contexte d'exécution est mise en œuvre pour valider cette plateforme.

La mobilité d'un agent est définie comme suit: son exécution est interrompue sur le nœud courant, appelé nœud source, ensuite les données représentant l'état de l'agent sont transférées du nœud source vers un nœud destination et enfin, arrivé au nœud destination, son exécution se poursuit là où elle avait été interrompue sur le nœud de départ. Cette opération, appelée migration du contexte d'exécution, est intégrée aux fonctionnalités d'un noyau temps réel, permettant ainsi la mobilité d'agents logiciels au sein d'une grappe de systèmes embarqués homogènes.

Les applications visées par le projet relèvent du domaine de l'informatique diffuse et plus particulièrement de son application à l'espace intelligent.

Mots-clés: agents mobiles, systèmes embarqués, noyau temps réel, informatique diffuse, espace intelligent, carte à puce RFID.

REMERCIEMENTS

Tout d'abord, j'exprime toute ma gratitude à mon directeur de recherche, le professeur Philippe Mabilleau, pour ses précieux conseils et orientations durant toute la réalisation de cette thèse.

Je tiens à remercier Abdoulaziz Souleiman Guelleh, ancien étudiant au doctorat de l'université de Montréal pour la relecture de cette thèse.

Je tiens à remercier Jacques Vanden-Abeele, professeur titulaire à la retraite et professeur associé au Département de génie électrique et de génie informatique de l'Université de Sherbrooke pour la lecture de cette thèse.

Je remercie Eduardo Romero, ancien étudiant à la maîtrise, pour son aide et la nombreuse discussion dans le domaine de systèmes embarqués.

Je remercie également tous ceux qui m'ont aidé de près ou de loin dans l'élaboration de ce travail.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LEXIQUE

API: Application Programming Interface.

Android: est un système d'exploitation en partie *Open Source* pour smartphones, PDA et terminaux mobiles.

Bluetooth: est une spécification de l'industrie des télécommunications. Elle utilise une technique radio courte distance destinée à simplifier les connexions entre les appareils électroniques.

BONDI: se compose:

- Spécifications des APIs: Un ensemble de pages HTML qui définissent la syntaxe et la sémantique des APIs BONDI;
- Spécifications de la politique de sécurité: Une description interopérable XML de la politique de sécurité qui définit l'accès que d'une application web particulière et widget aura à l'API de Bondi.
- Open source Reference Implementation (RI): les interfaces et les spécifications de sécurité.

BREW: Binary Runtime Environment for Wireless.

Bytecode Java: Code exécutable de Java.

CCE: Capture du contexte d'exécution.

CDMA: Code Division Multiple Access.

CORBA: Common Object Request Broker Architecture.

CPU: Central Processing Unit.

DM: Device Management.

drt: Débit réel de transmission.

EEA: Environnement d'exécution d'agents.

ENC: European Navigation Conference

GSM: Global System for Mobile Communications.

GNSS: Global Navigation Satellite System.

FIPA: Foundation for Physical Intelligent Agents.

IA: Intelligence artificielle.

IC: Infrastructure de communication.

IEEE: Institute of Electrical and Electronics Engineers.

IIOP: Internet Inter-ORB Protocol.

OMA: Open Mobile Alliance.
IrDA: Infrared Data Association.

JDK: Java Development Kit.
JEE: Java Enterprise Edition.

JME: Java Micro Edition.

JSE: Java Standard Edition.

JVM: Java Virtual Machine.

LCD: Liquid Crystal Display.

MAF: Mobile Agent Facility.

MASIF: Mobile Agent System Interoperability Facility.

MEMS: Microelectromechanical systems.

MMU: Memory Management Unit.

µC/MAS: Microcontroller Mobile Agent System

.NET: est le nom donné à un ensemble de produits et de technologies informatiques de l'entreprise Microsoft pour rendre des applications facilement portables sur Internet.

.NET CF: pour le .Net Compact Framework de Microsoft est une version du .NET.

NFC: Near Field Communication.

OBEX: OBject Exchange.

Objective-C: est un langage de programmation orienté objet réflexif. C'est une extension du C ANSI, comme le C++, mais qui se distingue de ce dernier par sa distribution dynamique des messages, son typage faible ou fort, son typage dynamique et son chargement dynamique.

OEM: Original Equipment Manufacturer

OMG: Object Management Group

OS: Operating System.

OSGi: Open Service Gateway Initiative.

PDA: Personal Digital Assistant.

RCE: Restauration du contexte d'exécution.

RISC: Reduced instruction set computing.

RF: Radio fréquence.

RFID: Radio Frequency Identification.

RMI: Remote Method Invocation.

RPC: Remote Procedure Calling.

RTOS: Real-Time Operating System.

SIM: Subscriber Identity Module.

SNP: Système de navigation pédestre

SOAP: Simple Object Access Protocol.

Symbian OS: est un système d'exploitation pour téléphones portables.

SSH: Secure Shell.

SSL: Secure Sockets Layer.

T: Période.

tce: Taille du contexte d'exécution.

TTL: Transistor-Transistor Logic.

ttrf: temps de transfert dans le réseau filaire.

ttrm: Temps de transmission de la trame.

UCT: Unité centrale de traitement.

UART: Universal Asynchronous Receiver/Transmitter.

UWB: Ultra Wide Band.

WiMAX: Worldwide Interoperability for Microwave Access.

XML: Extensible Markup Language

CHAPITRE 1
INTRODUCTION

1.1 Mise en contexte et problématique

Grâce à la miniaturisation, à la réduction des coûts de production et des prix unitaires, à l'augmentation continue de puissance de calcul et à l'intégration croissante des composants électroniques, il est envisagé de transformer tous les objets du monde physique (du plus petit au plus grand, du plus simple au plus complexe) en objets numériques, intelligents, autonomes et communicants [Shi et al., 2011] [Conti, 2007].

La perspective est celle d'un monde d'objets interconnectés dotés de capacités de communication, de détection et d'activation (réseaux de transport d'information sans fil, *RFID (radio frequency identification)*, *WSANs (Wireless sensor and actor networks)*, etc.) pour de nombreuses applications. Au-delà des multiples applications nouvelles, l'effet global consiste à enrichir qualitativement et quantitativement les interactions entre le monde de l'information/communication et le monde physique, autrement dit entre les bits et les atomes [Gershenfeld, 1999]. Ainsi, nous entrons dans l'ère de l'informatique omniprésente et sortons dans celle de l'ordinateur personnel (*PC*).

C'est dans ce contexte de l'informatique omniprésente que nous élaborons une solution de la mobilité d'agents mise en œuvre par des objets communicants (systèmes embarqués dotés de capteurs, d'actionneurs, de processeurs, d'interfaces RF, etc.) avec des contraintes fortes en termes de mémoire, de consommation d'énergies, de bande passante, etc. Le concept d'agents mobiles constitue une approche pouvant apporter des meilleures performances par rapport au modèle traditionnel client-serveur [Ismail et Hagimont, 1999] dans le contexte des applications réparties. En effet, la mobilité d'agents permet de déporter les traitements au plus près des données et éviter ainsi le coût de transferts de données et la latence de réseau.

1.2 Applications réparties

Il existe plusieurs paradigmes de programmation pour réaliser des applications réparties et le choix du paradigme dépend de l'application à laquelle il est destiné. Dans cette section, nous allons brièvement passer en revue ces différents paradigmes.

1.2.1 Modèle client-serveur

Plusieurs mécanismes d'interaction sont apparus pour mettre en œuvre le modèle client-serveur. Ils sont liés aux abstractions de haut niveau qui sont employées dans l'ingénierie des systèmes afin de réduire la complexité du procédé de construction.

Processus communicants

Le principe du modèle des processus communicants est assez trivial. Le système met à sa disposition des processus d'un service de messagerie via des ports de communication. Un processus peut envoyer ou recevoir un *message* sur un port de communication. Le système gère l'allocation des ports et le transport des messages via le réseau.

L'échange de messages entre les processus est un mécanisme puissant offrant beaucoup de souplesse pour la programmation des échanges. Néanmoins, le programmeur doit prendre en compte tous les problèmes liés aux techniques de communication comme la gestion des erreurs de transmission et des connexions, la construction et l'interprétation des messages, et la synchronisation entre les intervenants. Les systèmes répartis comme Mach [Milojicic, 1993] et Chorus [Rozier, 1992] sont des exemples qui utilisent cette approche.

Appel de procédure à distance

L'appel de procédure est un mécanisme de structuration des programmes permettant le transfert du contrôle et des données entre les différentes fonctions d'une application. L'appel de procédure à distance (en anglais *Remote Procedure Call, RPC*) [Birell et Nelson, 1984] est une extension de ce mécanisme pour les applications réparties fournissant une structuration

uniforme des activités locales ou distantes. C'est un mécanisme de plus haut niveau que l'échange de message facilitant la construction des applications réparties.

Le principe d'interaction est synchrone. La machine cliente appelle une procédure qui est sur la machine serveur et attend la réponse. L'environnement appelant est suspendu durant la transmission des paramètres, l'exécution de la procédure et la transmission des résultats. La figure 1.1 illustre le principe de fonctionnement qui utilise le concept du talon. Un talon, en anglais stub, est un programme chargé de représenter localement une activité distante afin de cacher la répartition. Le talon client représente le serveur sur le site appelant, il gère l'emballage et l'attente liés à l'exécution de la procédure. Systématiquement, le talon serveur représente le client sur le site de serveur, il gère l'emballage et la prise en compte des demandes d'exécution de la procédure.

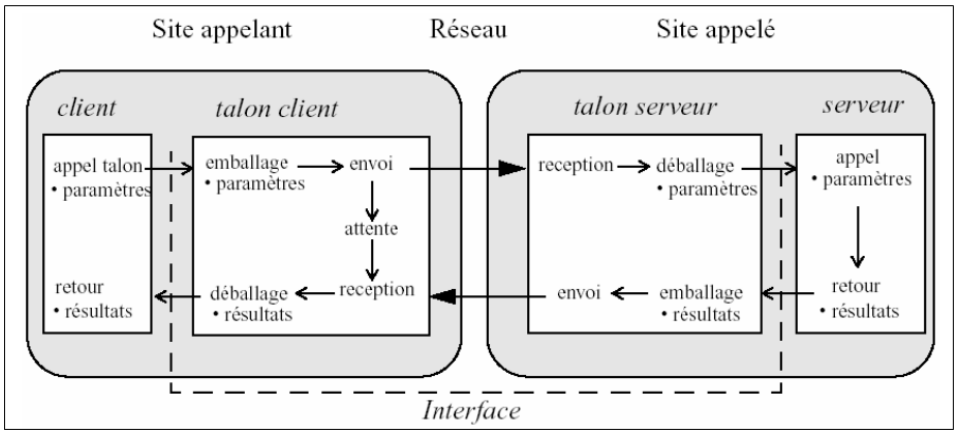


Figure 1.1 Principe de l'appel de procédure à distance

L'utilisation du RPC se fait à travers un langage de description d'interface IDL *(Interface Description Language)* et un générateur de talons. À partir d'une description unique de l'interface d'une procédure, on crée deux talons dans des langages pouvant être différents. Ces programmes sont ensuite compilés et liés séparément à l'application cliente et à l'application serveur. Un service s'exécute sur chaque machine du réseau. Ce service est chargé de recevoir les appels RPC venant des talons clients pour les adresser au talon serveur correspondant à la procédure. Lorsque qu'une application démarre, chaque serveur présent dans l'application s'initialise en s'enregistrant auprès du service pour se mettre en attente

d'une demande d'un client. Ainsi, chaque appel de procédure d'un client sera propagé jusqu'à la procédure correspondante du serveur.

L'appel de procédure à distance est très utilisé pour la construction d'applications réparties. Néanmoins, il ne permet pas de passer des paramètres par référence autre que par l'utilisation d'un article: il est possible de simuler le passage par référence en cachant la transmission par valeur. À l'entrée de la procédure, le système dépose une copie du paramètre et gère la recopie de celui-ci hors du retour. Cet artifice pose des problèmes de cohérence de copies multiples s'il existe des accès concurrents à ce paramètre. L'exemple de la double incrémentation [Tanenbaum, 1989] illustre une exécution conduisant à une erreur. La sémantique du passage de paramètre par référence n'est pas conservée dans l'appel de procédure à distance. Tous les paramètres emballés dans le message sont inévitablement passés par valeur, il n'est pas possible d'accéder à un paramètre par un pointeur à travers le réseau.

Appel d'objet à distance

Un objet est une unité de structuration qui regroupe des données et des méthodes permettant de manipuler ces données. Les systèmes à objet proposent directement cette abstraction aux applications. Le modèle d'exécution est fondé sur le mécanisme d'appel de méthode sur un objet à partir de sa référence dans le système. Pour réaliser ce modèle, le système fournit un mécanisme de liaison dynamique de l'objet dans le contexte de l'application appelante. L'appel d'objets à distance est une extension de ce modèle à un environnement réparti permettant de passer des paramètres par références.

Le principe d'appel d'objet à distance est fondé sur trois éléments: un schéma de désignation commun pour tous les objets, un mécanisme de liaison dynamique des objets dans le contexte d'une activité et l'utilisation d'un représentant de l'activité sur le site cible.

La figure 1.2 illustre ce principe. Une activité réalise l'appel d'une méthode d'un objet à partir de sa référence avec des paramètres. Le système détecte que l'objet référencé est distant grâce au service de désignation des objets et déclenche la procédure d'appel à distance. La

référence, le nom de la méthode et les paramètres sont emballés et transmis au site où se trouve l'objet. On notera que les paramètres peuvent contenir des références à d'autres objets. Le système sur le site cible reçoit cette demande et instancie un représentant de l'activité. Cette nouvelle activité effectue une liaison dynamique de l'objet désigné par la référence dans son contexte d'exécution. Elle réalise ensuite l'appel local de l'objet par l'appel de la méthode avec les paramètres. Cette exécution peut déclencher d'autres appels d'objet local ou distant. Lorsque chaque appel distant se termine, le représentant est détruit et l'activité initiatrice reçoit les résultats et peut continuer son exécution.

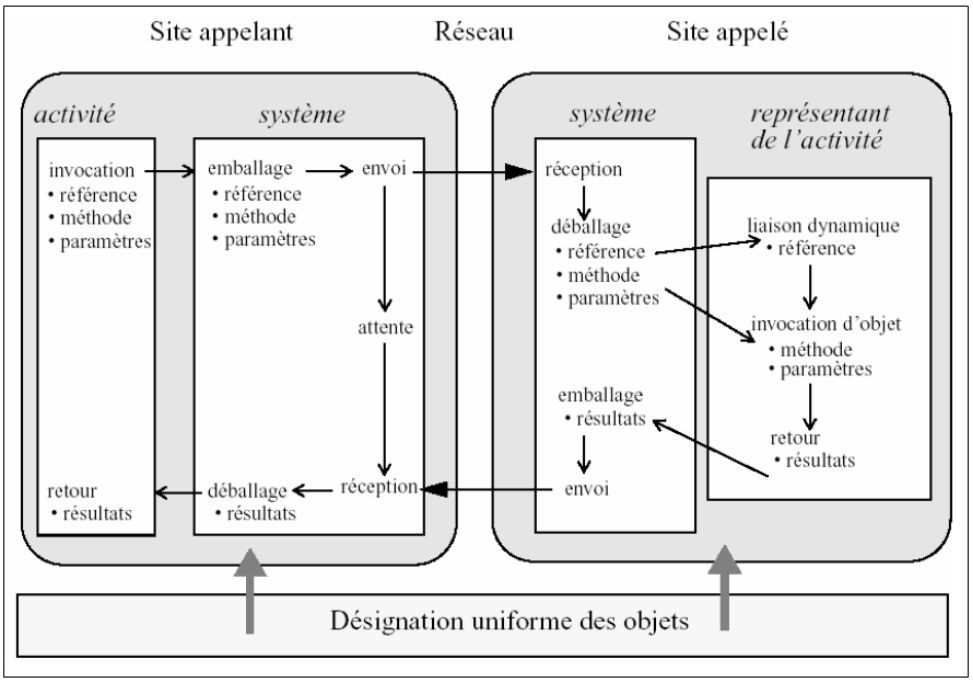


Figure 1.2 Principe de l'appel d'objet à distance

CORBA

Object Management Group (OMG) a défini *CORBA (Common Object Request Broker Architecture)*: un standard de communication entre les objets. *OMG* est un consortium international regroupant des fournisseurs des systèmes et des utilisateurs finaux. L'objectif de ce groupe est de faire émerger des standards pour l'intégration d'applications réparties hétérogènes à partir des technologies orientées objet. Ainsi les concepts-clés mis en avant sont la réutilisation, l'interopérabilité et la portabilité de composants logiciels. Ce standard CORBA, qui est un bus d'objets répartis, offre un support d'exécution masquant les couches

techniques d'un système réparti (système d'exploitation, processeur et réseau) et il prend en charge les communications entre les composants logiciels formant les applications réparties hétérogènes. *OMG* ne produit pas de code pour le bus *CORBA*, il ne fournit que des spécifications. L'implémentation (le passage de la spécification au programme) est laissée aux éditeurs, constructeurs, revendeurs et les utilisateurs finaux.

Le bus *CORBA* propose un modèle orienté objet client-serveur entre les applications réparties. Chaque application peut exporter certaines de ses fonctionnalités sous la forme d'objets *CORBA*. Les interactions entre les applications sont alors matérialisées par des appels à distance des méthodes des objets. La notion client-serveur intervient uniquement lors de l'utilisation d'un objet: l'application implantant l'objet est le serveur, l'application utilisant l'objet est le client bien qu'une application peut tout à fait être à la fois cliente et serveur. L'appel de méthode distante permet de structurer les programmes selon un modèle familier au programmeur. Les appels sont synchrones et la localisation de la méthode appelée est transparente au programmeur.

1.2.2 Du code mobile aux agents mobiles

Comme nous le montre la figure 1.3, une application mobile *(*processus ou agent*)* est composée de trois éléments: un *segment de code*, un *état d'exécution* (une pile et le bloc de contrôle) et des *données* [Fuggetta et al., 1998]. Pour permettre une efficacité accrue, il faut regrouper les trois éléments sur la même machine.

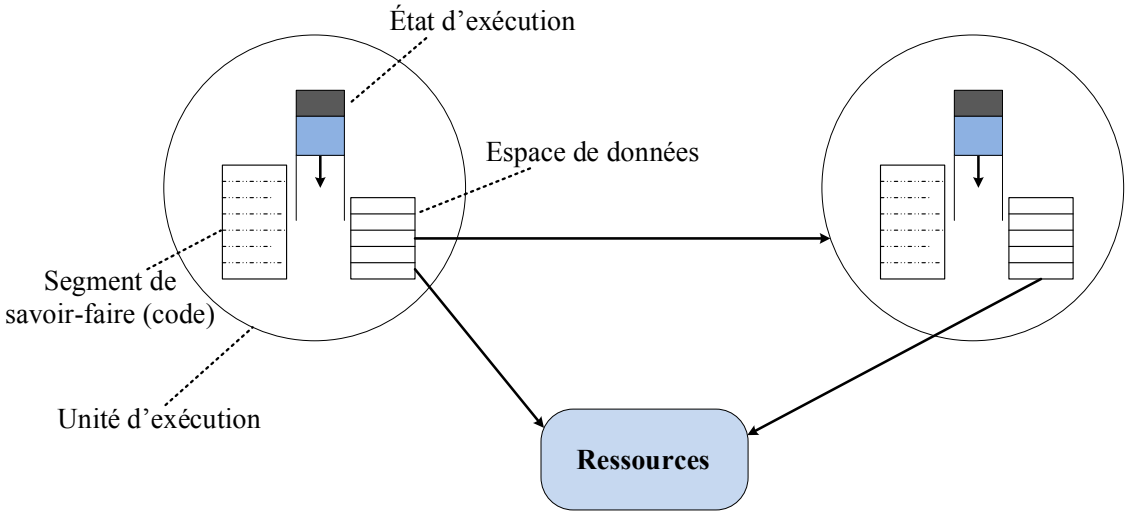


Figure 1.3 Environnement d'exécution réparti

Dans une application, il existe trois degrés de mobilité comme nous le montre la figure 1.4:

- La mobilité de savoir-faire (code): l'envoi et récupération du savoir-faire;
- La mobilité faible: le savoir-faire et les données courantes;
- La mobilité forte: le savoir-faire, les données courantes et l'état d'exécution.

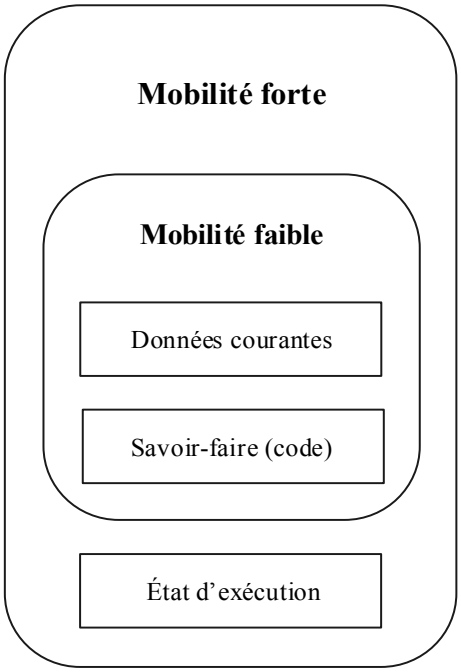


Figure 1.4 Degrés de mobilité

Mobilité faible

La mobilité faible consiste à transférer l'exécution de l'application d'une machine source vers une machine destination, en passant par l'interruption de l'exécution de l'application sur le site source. Ensuite, le code et les données courantes de l'application du site source sont transférés vers le site destination. Enfin, arrivée sur le site destination, l'application mobile reprend son exécution depuis le début tout en possédant les valeurs mises à jour de ses données.

Mobilité forte

En plus des informations prises en compte par la mobilité faible (code + données courantes), la mobilité forte prend également en compte l'état courant de l'exécution de l'application. Ainsi, une application fortement mobile qui se déplace au cours de son exécution d'un site source vers une destination peut reprendre son exécution à partir du point où elle a été interrompue sur le site de départ. La mobilité d'une application se traduit par l'interruption de l'exécution de l'application sur le site source. Le code, les données courantes utilisées et l'état courant de l'exécution de l'application du site source sont ensuite transférés vers le site destination. Enfin, arrivée sur le site destination, l'application mobile poursuit son exécution là où elle a été interrompue sur le site de départ.

Envoi du savoir-faire (code)

Dans l'approche de l'envoi du savoir-faire (code), plus communément appelé évaluation distante, c'est le site client qui dispose le savoir-faire propre du service à réaliser [Carzaniga et al., 2007] [Cubat dit cros, 2005] mais les ressources et l'unité d'exécution se trouvent sur le serveur. Dans cette approche, un site client envoie un code à un autre site. Le site récepteur exécute le code reçu qui peut contenir des données. Éventuellement, une interaction additionnelle délivre ensuite les résultats du service au site émetteur. Tel qu'illustré à la figure 1.5, l'unité d'exécution et les ressources sont fixes, seul le code est mobile. Un exemple d'utilisation de l'évaluation distante est le cas d'un client envoyant directement le code au serveur, comme la commande *rsh (Remote Shell)* du système Unix qui permet d'exécuter un script sur une machine distante.

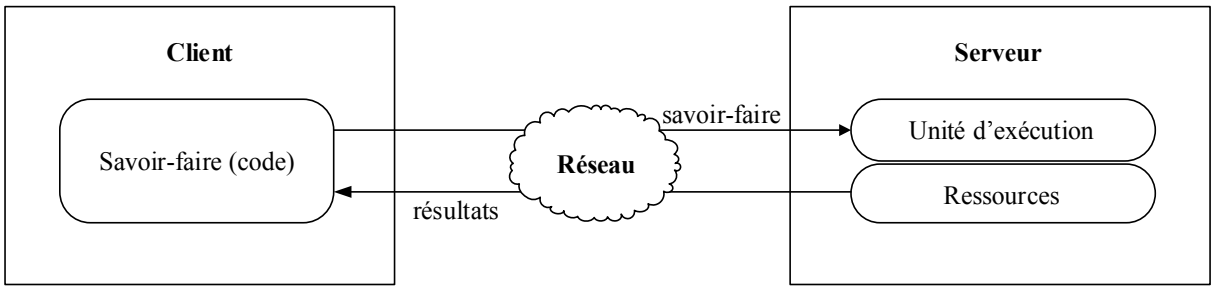


Figure 1.5 Modèle d'envoi du savoir-faire, appelé aussi évaluation distante

Récupération du savoir-faire (code)

Dans cette approche, appelé aussi code à la demande, le client dispose de l'unité d'exécution et des ressources mais pas du savoir-faire qui va être récupéré auprès du serveur [Carzaniga et al., 2007] [Cubat dit cros, 2005]. La récupération du savoir-faire [Carzaniga et al., 2007] ressemble à du téléchargement. Le site client interagit avec le site distant afin de récupérer le code nécessaire à la réalisation d'un service. Le site distant fournit le code du service à exécuter sur le site client tel qu'illustré à la figure 1.6. L'applet Java est un exemple qui utilise cette approche.

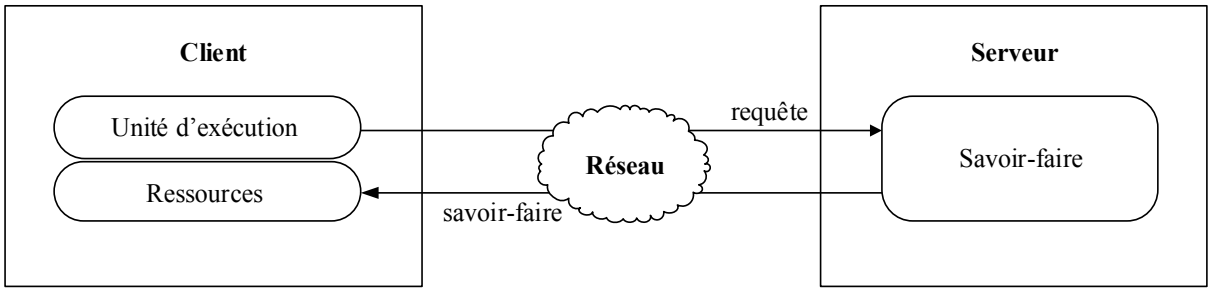


Figure 1.6 Modèle de récupération du savoir-faire, appelé aussi code à la demande

Migration de processus

La migration de processus est typiquement implémentée au niveau de systèmes d'exploitation. Elle utilise la *migration forte* puisqu'on déplace le savoir-faire, les données courantes et l'état d'exécution. Comme la migration de processus est initiée par le système, on parle alors de *migration réactive*. Lorsqu'un processus exprime ses besoins en ressources, comme l'accès à un fichier ou à du temps processeur, le système va placer le processus demandeur sur la machine qui répond le mieux à ses besoins.

Les premiers travaux sur la migration de processus ont été réalisés sur des systèmes homogènes. Ainsi, un processus s'exécutant sur une machine ne peut se déplacer que vers une machine de même architecture physique et de même système d'exploitation.

Après la migration de processus dans les systèmes homogènes, les recherches menées dans le domaine se sont ensuite intéressées à la migration de processus dans des systèmes hétérogènes. La complexité d'une telle migration dépend en particulier du fait que la représentation de l'état d'exécution d'un processus soit dépendante ou indépendante du système d'exécution.

Lorsque la représentation de l'état d'exécution d'un processus est indépendante de l'environnement d'exécution, la mise en œuvre de la mobilité du processus dans un système hétérogène est équivalente à sa mise en œuvre dans un système homogène. Pour illustrer cette approche, nous allons présenter une synthèse de l'approche Attardi [Attardi et al., 1988] où l'auteur définit une machine universelle d'abstraction dans le chapitre 2.

En revanche, si la représentation de l'état d'exécution d'une application est fortement dépendante de la plateforme sous-jacente et varie d'une plateforme à une autre, l'état d'exécution ne peut être directement transféré et utilisé par des plateformes hétérogènes. Pour résoudre ce problème, il faut procéder par des traductions entre les différents formats. Pour ce faire, deux approches sont envisageables.

La première approche consiste à traduire la représentation de l'état de l'exécution de l'application mobile du format de la plateforme source vers le format de la plateforme destination. Dans ce cas, il faut autant de traducteurs qu'il existe de couples de plateformes différentes. De plus, lorsqu'une nouvelle plateforme doit être prise en compte par le mécanisme de mobilité, il faut construire 2N nouveaux traducteurs, N étant le nombre de plateformes déjà prises en compte par le mécanisme de mobilité [Bouchenak, 2001]. Cette approche est proposée par Shub [Shub, 1990] pour mettre en œuvre un mécanisme de migration de processus.

La seconde approche, qui est une autre alternative à la traduction directe des formats de deux plateformes, est de se baser sur un format intermédiaire pour représenter l'état d'exécution d'une application mobile. La représentation de l'état d'exécution de l'application mobile est traduite du format de la plateforme source vers le format intermédiaire puis au format de la plateforme destination. Cette approche nécessite un nouveau type de plateforme, elle présente l'avantage de n'exiger la construction que deux traducteurs: un traducteur du format de la nouvelle plateforme vers le format intermédiaire et un traducteur inverse. Le système réparti Stardust [Cabillic et Puaut, 1997] et le système de migration Tui [Smith et Hutchinson, 1997] fournissent un mécanisme de la migration de processus qui suit une telle approche. Pour illustrer cette approche, nous allons présenter une synthèse du système Tui dans le chapitre 2.

Agents mobiles

Les agents mobiles sont la conjonction de deux domaines de recherches distincts: les agents venant de l'intelligence artificielle avec les systèmes multiagents [Ferber, 1999] et les systèmes distribués avec la migration de processus [Milojicic, 2000]. Les agents sont dits *mobiles* lorsqu'ils sont capables de migrer entre différents nœuds physiques d'un réseau. Cette principale caractéristique de mobilité n'est pas tributaire de l'intelligence dont les agents pourraient être dotés. Dans les systèmes courants, les agents se prévalent très peu des abstractions développées en intelligence artificiel. Ils empruntent surtout à la migration de processus qui consiste en transfert de processus entre les ordinateurs [Pierre, 2011].

Les agents mobiles viennent d'enrichir les notions de code mobile et d'objet mobile. Ils sont maintenant implémentés en langage interprété qui supporte le code mobile et sont indépendants des systèmes d'exploitation. Le modèle à base d'agents mobiles peut être vu comme une généralisation de la migration de processus où le déplacement est à l'initiative du code d'agent. Nous parlons alors d'une *migration proactive*. La figure 1.7 montre un modèle d'exécution d'agents mobiles.

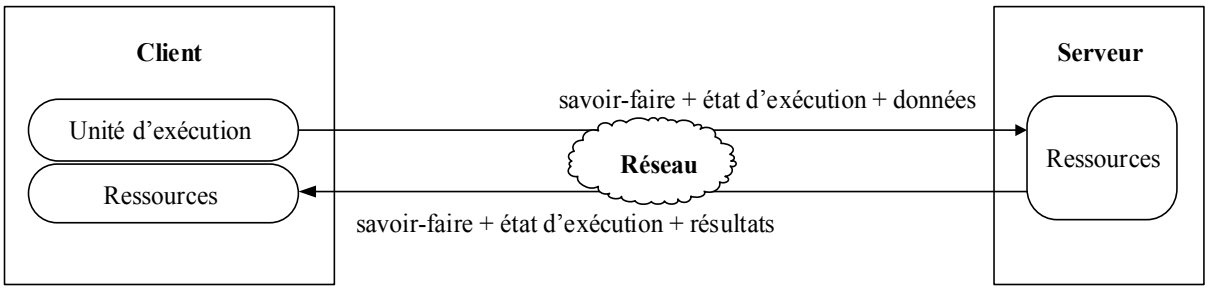


Figure 1.7 Modèle d'exécution d'agents mobiles

1.2.3 Avantages et limites de différents degrés de mobilité

Mobilité faible

Les agents mobiles qui suivent le modèle de mobilité faible ne conservent pas de connaissance des données traitées ou des actions exécutées dans des hôtes précédemment visités et, par conséquent, ils peuvent seulement mettre en œuvre des tâches dans lesquelles ces informations ne sont pas exigées.

Une utilisation pratique de la mobilité faible est la décentralisation dynamique des tâches de gestion de réseaux qui sont par ailleurs réalisées de manière centralisée. L'agent est délégué d'une partie de la responsabilité de la gestion et intégrera des fonctionnalités telles que des procédures visant à la compression des données ou le regroupement.

Un exemple typique montrant les principaux avantages de la mobilité faible est le cas où une station de gestion doit rechercher une valeur unique dans une table qui se trouve dans une structure de données sur une machine distante. Traditionnellement dans la gestion SNMP (abréviation pour *Simple Network Management Protocol*) [Bohoris et al,. 2000], la table entière doit être transférée à la station de gestion. Ainsi, le transfert de grandes tables occasionneront de lourd trafic inutile sur le réseau et entraîneront une surcharge de calcul sur la station de gestion.

Mobilité forte

La mobilité forte est utilisée dans plusieurs domaines et nous pouvons citer la répartition dynamique de charge, la tolérance aux fautes et l'administration du système et réseau. Nous allons présenter de façon détaillée les différents domaines d'applications de la mobilité forte à la section 2.2 du chapitre 2.

Dans le domaine de l'administration/gestion de réseau, les agents avec la mobilité forte sont en mesure d'accéder et de traiter les données à partir des éléments du réseau mais peuvent aussi rassembler de l'information et de la préserver lors de la migration. Cette fonction permet la mise en œuvre des tâches plus complexes dans lesquelles les opérations de l'agent dépendent de données recueillies chez des hôtes visités précédemment. Dans la gestion du réseau, la mobilité forte est plus adaptée à des tâches de configuration et de tâches intensives de données impliquant le regroupement de données d'éléments de réseau fortement distribués et l'analyse de ces données durant l'exécution. Un exemple type est le cas d'une tâche impliquant la collecte d'informations sur un grand nombre d'éléments de réseau.

Dans un cadre traditionnel basé sur un système SNMP, la station de gestion doit interroger chaque élément pour rassembler les informations de performance exigées avant qu'elle ne puisse produire un taux d'utilisation utile. La gestion OSI (abréviation pour *Open Systems Interconnection*) offre un mécanisme pour obtenir les taux d'utilisation, mais comme ceci est fait localement à l'élément de réseau, elle exige toujours un nouveau regroupement de ces informations à la station de gestion. Avec la mobilité forte, l'agent pourra préserver les taux d'utilisation d'éléments précédemment visités et pourra alors effectuer un autre niveau de regroupement de données indépendamment du gestionnaire.

Le principal inconvénient associé à la mobilité forte est la taille de l'agent. Les agents avec mobilité forte ont tendance à incorporer plus d'information. La taille de l'agent peut varier considérablement en fonction de la quantité d'information qui doit être préservée lors de la migration. Il est donc important de concevoir les agents de manière à limiter leur taille.

1.3 Contributions

L'étude de la mobilité des agents n'est pas un problème nouveau mais l'application du concept dans le contexte des systèmes embarqués est nouvelle. Un système embarqué est un système électronique piloté par un logiciel qui est complètement intégré au système qu'il contrôle. Contrairement aux systèmes destinés à l'usage général, les systèmes embarqués sont dédiés pour effectuer des tâches spécifiques. La caractéristique clé de systèmes embarqués reste le fait qu'ils sont dédiés pour fonctionner sur des machines avec des ressources très limitées (puissance du processeur, taille de mémoire, consommation d'énergie, etc.).

Bien que de nombreux travaux de recherche ont été effectués dans le domaine des agents mobiles incluant des thèses de doctorats [El falou, 2006] [Cubat dit cros, 2005] [Bouchenak, 2001] ainsi que des articles des journaux et des conférences [Carzaniga, et al., 2007] [Fuggetta et al., 1998], aucun n'a traité la problématique de la mobilité d'agents sur des systèmes embarqués de petite taille. Les solutions existantes de la mobilité d'agents reposent sur des systèmes d'exploitation utilisant plus de ressources. En effet, ces systèmes sont déployés sur des machines dotées d'un processeur plus rapide, d'une mémoire de grande capacité, d'un module matériel pour la gestion de mémoire, etc.

L'objectif de ce projet de recherche consiste, dans un premier temps, à proposer une solution aux problèmes de la mobilité d'agents dans le contexte de systèmes embarqués temps réel. Dans un deuxième temps, nous mettons en œuvre un exemple d'application exploitant notre solution dans le contexte d'informatique diffuse.

Dans la solution proposée, la mobilité d'un agent est définie comme suit: son exécution est interrompue sur le nœud courant, appelé nœud source, ensuite le code et les données représentant l'état de l'agent sont transférés du nœud source vers un nœud destination. Enfin, arrivés au nœud destination, l'exécution de l'agent se poursuit là où elle avait été interrompue sur le nœud de départ.

Le travail accompli dans cette thèse a essentiellement permis d'apporter des contributions aux niveaux système et application. Au niveau système, ce travail a apporté les contributions suivantes:

- ✓ D'abord, une méthode de migration d'agents dans le contexte de systèmes embarqués temps réel. En exploitant l'analogie qui existe entre le changement du contexte d'exécution de tâches par un noyau temps réel et la mobilité d'agents, nous avons conçu une méthode de migration d'agents. Il est pertinent de préciser qu'il s'agit d'une mobilité forte.
- ✓ La deuxième contribution de cette thèse est la définition d'une directive de migration des données. À l'aide d'un arbre binaire où chaque feuille spécifie la classe de stockage des variables ainsi que leur migration ou non avec l'agent, nous avons défini une directive de migration des données.
- ✓ La troisième contribution de ce travail concerne la spécification d'un format de transfert et une pile de protocole pour la migration d'agents à l'aide de la norme XML en conjonction avec celle Intel HEX.
- ✓ La quatrième contribution de cette thèse est l'introduction d'une nouvelle combinaison: le paradigme d'agents mobiles et la technologie de cartes à puce RFID. L'emploi des cartes à puce RFID comme moyen de transport d'agents apporte une indépendance (autonomie) par rapport à toute utilisation de réseau. Les données extraites de la carte à puce RFID contiennent toutes les informations nécessaires pour restaurer l'état d'exécution de l'agent dans son nouvel environnement d'accueil.

Au niveau système, ceci se traduit concrètement par des services de mobilité basés sur des mécanismes de plus bas niveau, des mécanismes de capture et de restauration de l'état d'exécution d'agents. Ces services sont intégrés aux fonctionnalités d'un noyau temps réel permettant ainsi la mobilité d'agents logiciels au sein d'une grappe de systèmes embarqués homogènes. L'originalité de cette solution de migration est qu'elle peut être mise en œuvre dans n'importe quel système multitâche puisque nous exploitons des mécanismes de bas niveau qui existent déjà dans ces systèmes.

Au niveau application, notre contribution consiste à la mise en œuvre d'un système pédestre non seulement pour vérifier et valider le bon fonctionnement mais également montrer l'utilité de notre solution de la mobilité forte d'agents dans le contexte de systèmes embarqués temps réel.

1.4 Démarche suivie

La démarche que nous adoptons pour aborder le problème de la mobilité d'agents dans le contexte de systèmes embarqués temps réel est composée de trois étapes:

1) L'étude des travaux antérieurs depuis la migration de processus jusqu'aux plateformes d'agents mobiles;
2) L'élaboration d'une solution de la mobilité qui se traduit par le développement d'une plateforme d'agents mobiles pour systèmes embarqués reposant sur l'extension d'un noyau temps réel;
3) La réalisation d'un exemple d'application exploitant notre solution de la mobilité d'agents en l'occurrence un système de navigation pédestre.

1.5 Structure du document

Ce document est organisé en sept chapitres dont le deuxième et le troisième présentent le thème de recherche abordé dans cette thèse. Le deuxième chapitre présente l'étude des travaux effectués dans le domaine de la migration de processus qui est la base de systèmes d'agents mobiles. Dans ce chapitre, nous commençons par définir le concept de processus afin d'identifier ses différents éléments qui le constitue. Par la suite, nous présentons les mécanismes de migration de processus dans des machines homogènes et hétérogènes ainsi que des évaluations des performances.

Dans le troisième chapitre, nous étudions les agents mobiles et leurs spécificités. Premièrement, nous commençons par définir en mettant l'accent sur les caractéristiques propres aux agents mobiles. Deuxièmement, nous présentons les travaux sur la modélisation des systèmes à base d'agents mobiles. Troisièmement, nous décrivons les efforts de

standardisation des plateformes d'agents mobiles. Puis, nous exposons des exemples de plateformes supportant le modèle des agents mobiles et leurs domaines d'application. Nous présentons ensuite un bilan des avantages et inconvénients des agents mobiles. Enfin, nous finissons ce chapitre qui termine la revue des travaux dans le domaine de la mobilité des applications par une conclusion où nous nous positionnons par rapport aux autres recherches.
Dans le quatrième chapitre, nous décrivons les différents éléments de notre solution de la mobilité qui se traduit par le développement d'une plateforme d'agents mobiles pour systèmes embarqués reposant sur l'extension d'un noyau temps réel. Pour commencer, nous présentons la méthode de migration d'agents. Ensuite, nous exposons la directive de migration d'agents. Enfin, nous décrivons le format de transfert d'agents.

Dans le cinquième chapitre, nous exposons la mise en œuvre de la solution de mobilité proposée. Pour commencer, nous décrivons l'architecture de la bibliothèque de la plateforme d'agents mobiles pour systèmes embarqués. Nous présentons par la suite les primitives permettant la capture et la restauration du contexte d'exécution et des données courantes. Nous décrivons les services de communication. Puis, nous exposons l'évaluation de la plateforme d'agents mobiles en effectuant un certain nombre des tests sur différents réseaux.

Le sixième chapitre présente un exemple d'application exploitant notre solution de la mobilité d'agents en l'occurrence un système de navigation pédestre. Nous passons d'abord en revue les systèmes de navigation pédestre existants. Ensuite, nous décrivons notre système de navigation pédestre. Enfin, nous finissons par une comparaison de notre système de navigation avec des systèmes similaires.

Dans le septième chapitre, nous dressons les contributions de la thèse et ainsi que les perspectives des recherches futures.

CHAPITRE 2
MIGRATION DE PROCESSUS

Ce chapitre présente l'étude des travaux effectués dans le domaine de la migration de processus qui est la base de systèmes d'agents mobiles. Nous commençons par définir le concept de processus afin d'identifier ses différents éléments qui le constitue. Par la suite, nous abordons les mécanismes de migration de processus dans des machines homogènes et hétérogènes ainsi que les évaluations des performances.

2.1 Introduction

Dans cette section, nous identifions les différents éléments fondamentaux qui interviennent dans la migration de processus. Rappelons qu'un agent mobile est un processus où la migration est initiée par son code.

2.1.1 Concept de processus

Le concept de processus est fondamental dans un système d'exploitation. Un processus est l'image d'un programme en cours d'exécution en mémoire. Les processus sont les seuls "éléments actifs" du système. Nous n'envisageons pas le cas où un processus s'exécuterait sur plusieurs processeurs simultanément. L'exécution de processus se fait d'une façon séquentielle: les instructions qui la composent sont chargées dans le processeur puis exécutées les unes après les autres. Un processus est plus que le code d'un programme. Un processus est caractérisé à un instant donné par: les données, la pile, le tas, la valeur du compteur ordinal, le contenu des registres ainsi que l'état spécifique du système d'exploitation sous-jacent comme les paramètres liés pour traiter la mémoire et la gestion de fichiers. Un programme est une entité passive comme le contenu d'un fichier stocké sur le disque tandis qu'un processus est une entité active avec un compteur ordinal spécifiant d'adresse de la

prochaine instruction à exécuter et des ressources associées. Un processus est dynamique par opposition à un programme qui lui est statique.

Bien que deux processus puissent être associés au même programme, on les considère néanmoins comme deux séquences d'exécution séparées. Par exemple, plusieurs utilisateurs peuvent exécuter une copie du même programme. Chacune de celle-ci est un processus et, bien que le code de programme soit équivalent, les sections de données varieront. Il est aussi commun d'avoir un processus qui engendre plusieurs processus en s'exécutant. Comme le présente la figure 2.1, une structure d'un processus en mémoire est composée d'un segment de:

- La *pile* permettant de stocker les appels de fonctions avec leurs paramètres et leurs variables locales. Lors d'un retour de fonction, les paramètres et variables sont dépilés.
- Le *tas* réservé aux allocations dynamiques de mémoire.
- Les *Données* contenant les variables globales.
- Le *Code* correspondant aux instructions du programme à exécuter. Ce code est le fichier exécutable produit par le compilateur ou assembleur.

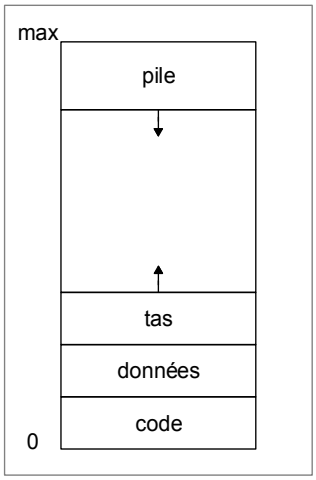


Figure 2.1 Structure d'un processus en mémoire virtuelle

État d'un processus

Un processus en exécution change d'état. L'état d'un processus est défini en partie par son activité courante. Chaque processus peut être dans un des états suivants [Silberschatz et al., 2012]:

- nouveau: le processus est en cours de création,
- en exécution: le processus s'exécute sur le processeur,
- en attente: le processus attend l'arrivé d'entrée/sortie ou d'événement,
- prêt: le processus attend pour être assigné au processeur,
- terminé: le processus a terminé son exécution.

Les noms des états sont arbitraires et varient entre les systèmes d'exploitation. Cependant, les états qu'ils représentent se trouvent sur tous les systèmes. Certains systèmes d'exploitation décrivent plus des états, mais l'importance est à un instant donné un seul processus peut être exécuté. Cependant, beaucoup de processus peuvent être prêts et en attente. La figure 2.2 illustre les différents états d'un processus.

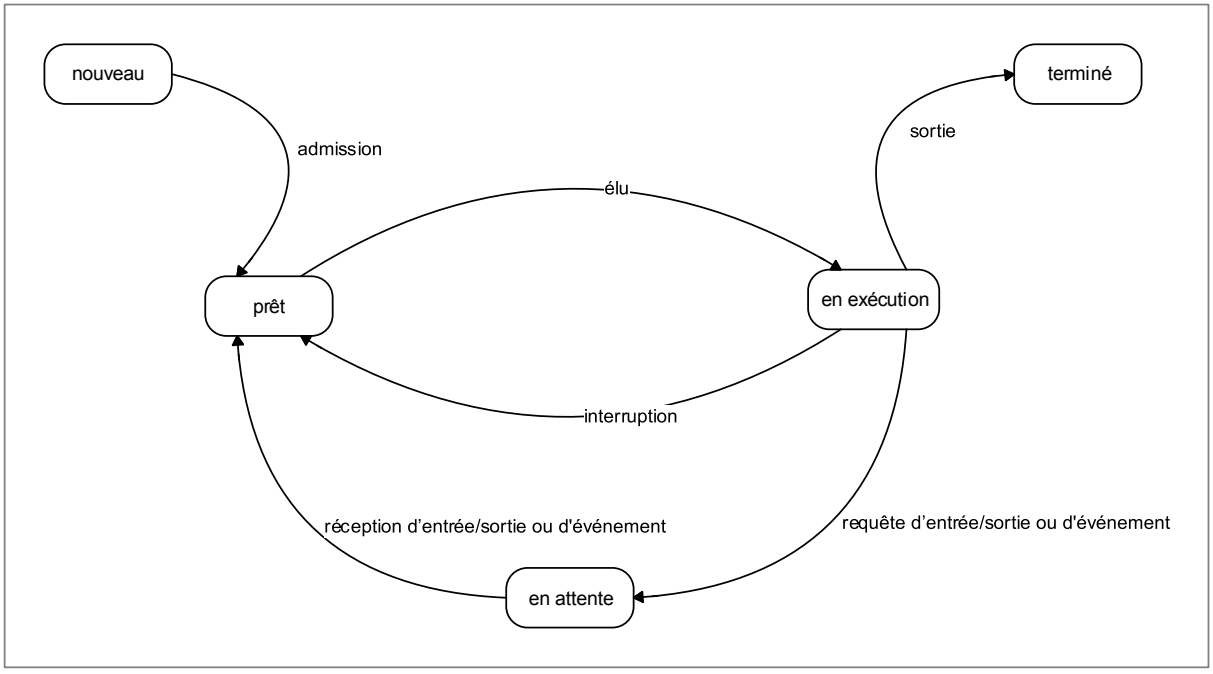


Figure 2.2 Diagramme de transition d'états d'un processus

Bloc de contrôle de processus

Chaque processus est représenté dans le système d'exploitation par un bloc de contrôle de processus (*process control block*), appelé aussi bloc de contrôle de tâche (*task control block*) tel que l'illustre la figure 2.3. Pour gérer les processus, le système d'exploitation sauvegarde plusieurs informations dans des structures de données. Le contenu d'un bloc de contrôle d'un processus (BCP) peut varier d'un système à l'autre. En générale, on trouve dans chaque processus les informations suivantes:

- L*'état de processus*: nouveau, prêt, en exécution, en attente, terminé, suspendu, etc.
- Le *numéro d'identification* unique est affecté à la création de chaque processus.
- Le *compteur ordinal*: Avant de pouvoir connaître le contenu d'une instruction, il faut aller la chercher en mémoire centrale. Et c'est justement le rôle d'un registre particulier communément appelé le compteur ordinal dont le contenu désigne l'adresse de l'instruction suivante à exécuter avant de pouvoir la lire en mémoire.
- Les *registres*: Le nombre et le type de registres varient dépendamment de l'architecture de chaque ordinateur. Ils comportent des accumulateurs, des registres d'index, des pointeurs de piles et des registres généraux. Le compteur ordinal et l'état de processus sont sauvegardés lorsque le processus quitte l'état "en exécution" et seront rechargés une fois que le processus revient à son état "en exécution".
- L*'ordonnancement*: Les informations concernant les files d'attente, le numéro de priorité, etc.
- La *mémoire*: Les informations relatives à la quantité mémoire allouée au processus et les pointeurs vers ces zones mémoires.
- La *comptabilisation*: Les informations qui définissent le temps d'exécution alloué pour chaque processus.
- L*'état des opérations d'entrées/sorties*: les périphériques alloués, l'opération en attente, les fichiers ouverts, etc.

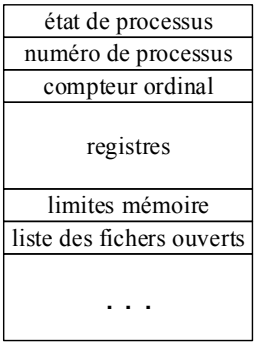


Figure 2.3 Bloc de contrôle de processus

2.1.2 Processus et mémoire

Chaque processus dispose d'un espace de mémoire. Après l'unité centrale de traitement (UCT), le second élément fondamental de tout ordinateur est la mémoire. Idéalement, la mémoire devrait satisfaire les trois critères [Tanenbaum, 2008]:

1. une mémoire extrêmement rapide (plus rapide que le temps d'exécution d'une instruction de façon à ce que le processeur ne soit pas freiné par les accès mémoire),
2. un espace mémoire de grande taille disponible,
3. une mémoire peu coûteuse.

Comme aucune des technologies courantes ne satisfait ces trois critères, la mémoire des ordinateurs actuels est organisée en plusieurs couches qui réalisent ainsi une hiérarchie de mémoire. Le but principal de cette hiérarchie est de simuler l'existence d'une mémoire rapide de grande capacité. Dans cette hiérarchie, les mémoires les plus proches du processeur sont les plus rapides mais aussi les plus petites en capacité. En revanche, les mémoires les plus éloignées du processeur sont les plus lentes mais aussi les plus grandes en capacité. Comme le montre la figure 2.4 [Cazes et Dlacroix, 2011] [Tanenbaum, 2008], la mémorisation de l'information dans un ordinateur ne se fait pas en un lieu unique mais elle est organisée dans une hiérarchie de mémoires.

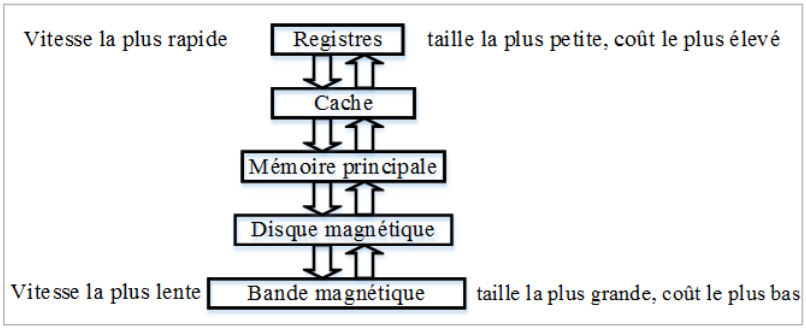


Figure 2.4 Hiérarchie de mémoires, vitesse, coût et taille.

Pour réaliser une mémoire rapide de grande capacité, la mémoire principale (la mémoire RAM) est découpée en blocs de taille fixe, appelées pages réelles ou cases. La mémoire des processus, mémoire virtuelle, est également découpée en blocs de même taille fixe, appelées pages. Pour faire la correspondance entre la page virtuelle et celle réelle, une table (une par processus) est gérée par le système d'exploitation. Un mécanisme de traduction des adresses virtuelles en adresses réelles associe à chaque numéro de page virtuelle le numéro de case réelle qui contient cette page, si elle existe. Cette traduction est assurée par un module matériel appelé unité de gestion de mémoire (en anglais *Memory Management Unit, MMU).*

2.2 Migration de processus

La migration de processus se définit par l'action de transférer un processus entre deux nœuds pendant son exécution qui sont reliés par un réseau de communications et n'utilisant pas une mémoire partagée. La migration de processus possède différents domaines d'applications et on peut en citer:

- *La répartition dynamique de charge* en migrant le processus en cours d'exécution d'un nœud source surchargé vers un nœud destination moins chargé.
- *La tolérance aux fautes* en migrant des processus d'un nœud source suspecté de devenir indisponible vers un nœud destination encore disponible. En plus, une panne qui survient sur la machine sur laquelle s'exécute une application provoque la terminaison prématurée de l'application et ainsi la perte de l'état d'exécution. Mais si des sauvegardes régulières de l'application ont été effectuées et stockées sur disque, l'application peut être reprise à partir de sa dernière sauvegarde [Bouchenak, 2001] [Huang et al., 1995].

- *L'administration de systèmes*: lors de reconfiguration du système, l'administration de systèmes répartis peut nécessiter la migration de certains services d'un nœud source vers un nœud destination. Ces déplacements de services peuvent être réalisés grâce au mécanisme de migration de processus. En outre, une administration de systèmes repartis nécessite parfois la réinitialisation de machines sans interrompre certains services (applications) en cours d'exécution sur ces machines. Ceci peut se faire en transférant ces services vers d'autres machines [Bouchenak, 2001] [Oueichek, 1996].
- *L'accès local*: une application peut accéder à des données se trouvant sur un nœud distant. Ce type de communications peut être coûteux dans le cas d'accès distants fréquents. Le déplacement de l'application vers le nœud où se trouvent les données utilisées transforme les accès à travers le réseau en accès local et réduit ainsi le temps d'exécution.

La migration de processus d'un nœud source vers un nœud destination consiste à:

- interrompre le processus s'exécutant sur le nœud source pour extraire son contexte d'exécution;
- transférer ce contexte vers le nœud destination;
- créer sur le nœud destination un nouveau processus auquel on affectera le contexte d'exécution transféré;
- mettre à jour les liens de communication avec les autres processus.

Le contexte d'exécution d'un processus est l'ensemble des informations qu'un processus peut consulter ou modifier.

2.2.1 Niveau de mise en œuvre de migration de processus

La migration de processus peut être appliquée à différents niveaux d'un système d'exploitation. Il en résulte que la migration de processus produit une performance, une tolérance aux fautes et une réutilisation différente. Les implémentations existantes de migration de processus sont faites au niveau: noyau ou application.

Migration de processus au niveau application

Les techniques de migration au niveau application supportent la migration de processus sans changer le noyau de système d'exploitation. Les mises en œuvre de migration au niveau application sont plus faciles mais ne permettent pas d'accéder à l'état de noyau d'où leur incapacité à faire migrer tout le processus.

Pour avoir accès à l'information de système, les mécanismes de migration de processus doivent passer de mode utilisateur en mode noyau comme nous le montre la figure 2.5 [Silberschatz et al., 2012]. Un exemple d'une mise en œuvre de migration de processus au niveau application est le système Condor [Litzkow et Livny, 1990] [Litzkow et al., 1988].

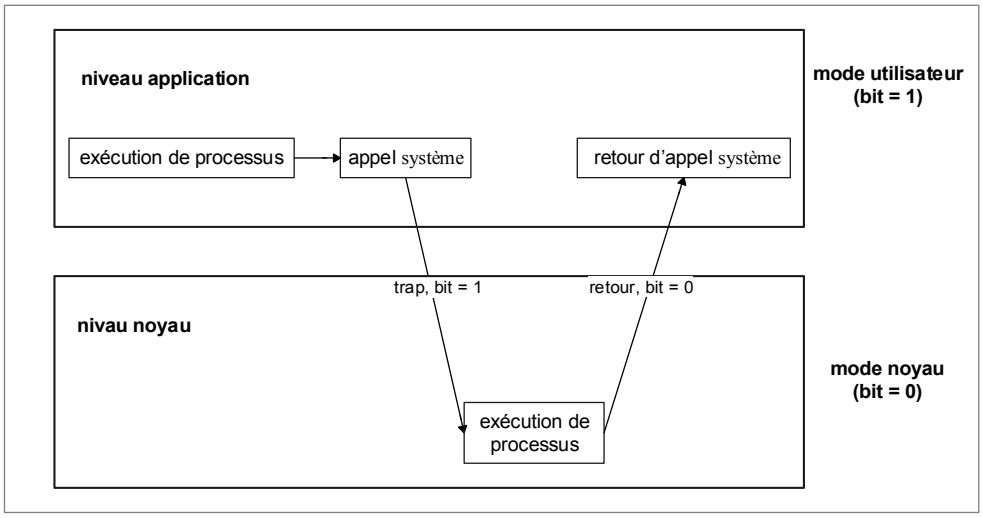


Figure 2.5 Transition de mode utilisateur en mode noyau

Migration de processus au niveau noyau

La migration de processus au niveau noyau implique des extensions du système d'exploitation sous-jacent qui mènent à une complexité supplémentaire. Le déploiement est plus difficile qu'avec une mise en œuvre de niveau application. L'accès direct et rapide à l'information de noyau permet le transfert de processus plus facilement et efficacement. La transparence à l'application est un autre avantage de migration de processus au niveau noyau. Les systèmes MOSIX [Barak et al., 1995] et Sprite [Douglis et Ousterhout, 1991] sont des exemples de mises en œuvre de migration de processus au niveau noyau.

2.3 Migration de processus dans des machines homogènes

Dans cette section, nous allons présenter les notions fondamentales de la migration de processus dans des machines homogènes: les algorithmes de migration, les métriques pour l'évaluation de temps de migration, les plateformes, l'espace d'adressage, etc.

L'espace d'adressage constitue la plus grande unité d'informations et reste donc l'élément qui a la plus grande influence sur la performance de migration de processus [Zayas, 1987b]. Différentes stratégies ont été proposées pour réduire le coût élevé de transfert de l'espace d'adressage. Ces stratégies jouent un rôle prédominant sur les caractéristiques de migration de processus telles que la performance, la complexité et la tolérance aux pannes. Tous les algorithmes de migration de processus existants ont ces fonctions de base suivantes:

- Décider de migrer le processus;
- Suspendre le processus à l'hôte source;
- Transférer l'état du processus à l'hôte destination;
- Reconstruire l'état du processus à l'hôte de destination;
- Reprendre l'exécution du processus à l'hôte de destination;
- Supprimer toutes les informations non nécessaires du processus de l'hôte source.

L'ordre et le degré de réalisation de chaque tâche varient entre les algorithmes de migrations. Le moment optimal pour migrer un processus particulier et l'hôte destinataire utilisé dépendent de la politique du système.

La principale différence entre les algorithmes de migration de processus dépend du moment où l'espace d'adressage d'un processus est transféré. L'espace d'adressage peut être transféré de deux manières:

(1) l'hôte source est envoyé tout d'un coup;

(2) il transfère en plusieurs parties sur demande.

Le premier procédé est contrôlé par l'hôte source et résulte d'un temps de migration initiale long. Dans la deuxième méthode, l'hôte de destination exige à distance une requête des pages

référencées de l'hôte source. Cette requête supplémentaire résulte d'un retard d'exécution du processus migré.

La plupart des algorithmes fondamentaux effectuent la migration de processus d'un hôte source directement vers un hôte de destination. Pour éviter des dépendances résiduelles, quelques algorithmes impliquent une troisième entité comme un serveur de fichiers. Notons que ces algorithmes utilisent la migration forte.

2.3.1 Algorithmes de migration de processus

Il existe huit algorithmes de base pour la migration de processus dans des machines homogènes [Noack, 2003]:

- L'algorithme de copie totale *(Total Copy Algorithm)*;
- L'algorithme de pré-copie *(Pre-Copy Algorithm)*;
- L'algorithme de page à la demande *(Demand Page Algorithm)*;
- L'algorithme de serveur de fichiers *(File Server Algorithm ou Flushing Algorithm)*;
- L'algorithme de gel libre *(Freeze Free Algorithm)*;
- L'algorithme de pré-copie de file en attente *(Queued Pre-Copy algorithm)*;
- L'algorithme de post-copie *(Post-Copy algorithm)*;
- L'algorithme de post-copie assistée *(Assisted Post-Copy algorithm)*.

L'algorithme doit aussi établir un ordre de transfert de différents composants de l'état de processus et essayer de réduire au minimum le nombre total de messages. Les différents composants de l'état de processus qui doivent être cueillis et transmis sont:

- L'information de contrôle de processus: la priorité, l'état, le numéro de processus, le numéro de parent de processus;
- L'état d'exécution: l'état de processeur, le pointeur de pile, le compteur ordinal, les registres d'état;
- L'espace d'adressage: la mémoire virtuelle entière du processus;
- Le lien de contrôle d'information et les messages de mémoire tampon;

- L'information de fichier: les descripteurs de fichier et les blocs de fichier mis en antémémoire.

Il existe plusieurs métriques importantes pour évaluer le temps de migration de processus. La figure 2.6 donne un modèle simplifié de ce temps qui est composé [Noack, 2003]:

- Le *temps de migration initiale* décrit le temps passé depuis la demande de la migration jusqu'à ce que le processus commence son exécution sur le nœud de destination.
- Le *temps de migration totale* est le temps passé depuis la demande de la migration jusqu'à la fin du transfert complet du processus. Les algorithmes de migration de processus avec des dépendances résiduelles n'ont pas de temps de migration totale. En effet, certaines parties de l'information de processus résident toujours à l'ancien nœud donc la migration ne finit jamais complètement.
- Le *temps mort* se réfère à la période où le processus de migration est suspendu. Pendant ce temps, l'état de processus est transféré au nœud destination selon l'algorithme de migration employé. Le temps mort est particulièrement critique pour les processus qui communiquent avec d'autres processus parce qu'ils ne peuvent pas recevoir des messages pendant ce temps.
- Le *temps de préparation* fait référence au transfert de plusieurs parties de l'information de processus, habituellement l'espace d'adressage avant la migration réelle.
- Le *temps post-traitement* est la période après la reprise d'exécution de processus dans le nœud destination pour éliminer, par exemple, les dépendances résiduelles de l'hôte source en expédiant toutes les données restantes du processus au nœud de destination. Les dépendances résiduelles arrivent quand l'information de processus reste à l'hôte source après la migration. Ainsi, le processus migré dépend toujours de son ancien nœud. Si l'hôte source ou la connexion de réseau tombe en panne, l'exécution du processus échouera à cause de l'information manquante.

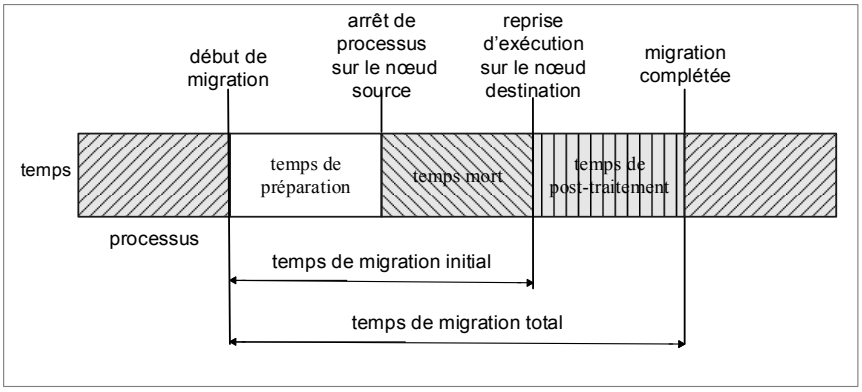


Figure 2.6 Périodes de migration de processus

Algorithme de copie totale

L'algorithme de copie totale est le premier et reste le plus largement utilisé pour la migration de processus. Il est utilisé dans les systèmes tels que Locus [Smith, 1995], Amoeba [Steketee et al., 1996] et Charlotte [Finkel et Artsy, 1989]. Le but de cet algorithme consiste à suspendre le processus, à transférer toute l'information d'état et ensuite à reprendre son exécution sur le nœud destination. D'une manière générale, l'algorithme de copie totale de migration de processus suit les étapes suivantes:

- Suspendre le processus de l'hôte source;
- L'hôte source envoie une requête d'approbation de migration à l'hôte destination;
- L'hôte destination répond par un message d'acceptation ou de rejet;
- Si la migration est acceptée, l'hôte source:
    - transfère le contrôle de processus et l'état d'exécution;
    - envoie les liens de communication et les messages de mémoire tampon;
    - transfère les descripteurs de fichier et les blocs de mémoire cache de fichiers modifiés;
    - envoie les pages de code, le tas et la pile;
    - demande à l'hôte destinataire de reprendre l'exécution de processus;
    - supprime le processus.

L'algorithme de copie totale a un défaut principal: le temps mort est long puisque le transfert du processus complet se fait seulement à cette période (voir la figure 2.7). Tout message qui est envoyé au processus durant ce temps de transfert est bloqué. Ceci augmente radicalement

l'apparence d'échecs de communication parce que le processus ne peut pas recevoir des messages pendant ce temps. L'avantage de cet algorithme est sa simplicité. Il élimine les dépendances résiduelles en expédiant toute l'information à l'hôte destinataire et peut ainsi supporter la tolérance aux pannes de processeur. La figure 2.8 présente l'algorithme de copie totale de migration de processus.

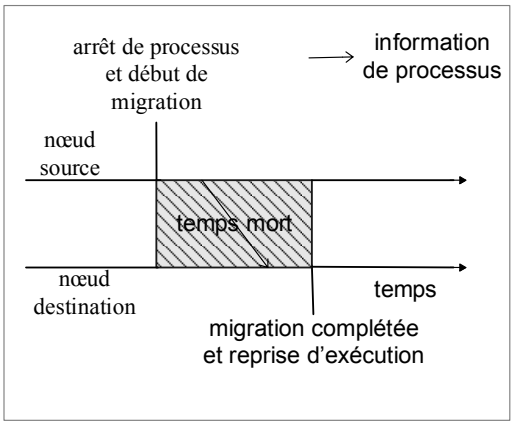


Figure 2.7 Période de transfert de l'algorithme de copie totale (voir la définition à la page 28)

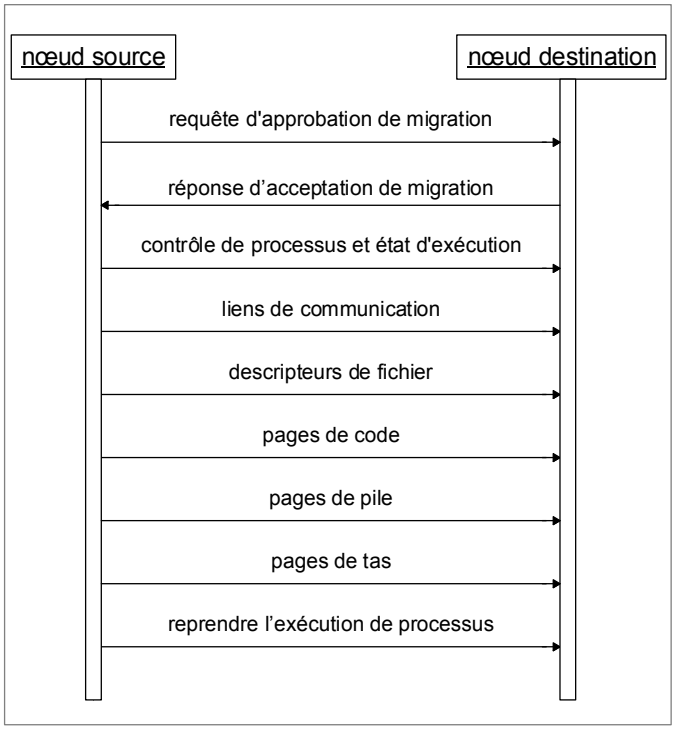


Figure 2.8 Algorithme de copie totale

Algorithme de pré-copie

L'algorithme de pré-copie a été inventé par Theimer [Theimer et al., 1985] pour le système V. Cette technique optimise le transfert de l'espace d'adresse virtuelle en permettant que le processus continue son exécution sur son nœud source pendant sa migration. Une fois le transfert accepté, le nœud source envoie les pages de code, de tas et de la pile au nœud destination. Si le nombre total de pages modifiées envoyées est plus grand qu'une limite prédéterminée, le processus continue son exécution et les pages nouvellement modifiées sont retransmises. Sinon, le processus est suspendu et toute l'information d'état est transférée.

L'algorithme de pré-copie réduit le temps de migration en limitant le nombre de pages de l'espace d'adresse virtuelle transmis pendant la suspension du processus. Cependant, le coût total augmente si les pages continuent à être modifiées pendant que le processus s'exécute sur le nœud source d'où le besoin de les retransmettre. Ceci peut prendre plusieurs envois jusqu'à ce que le nombre de pages modifiées soit inférieur à la limite prédéterminée. La figure 2.9 présente les périodes de migration de processus de l'algorithme de pré-copie.

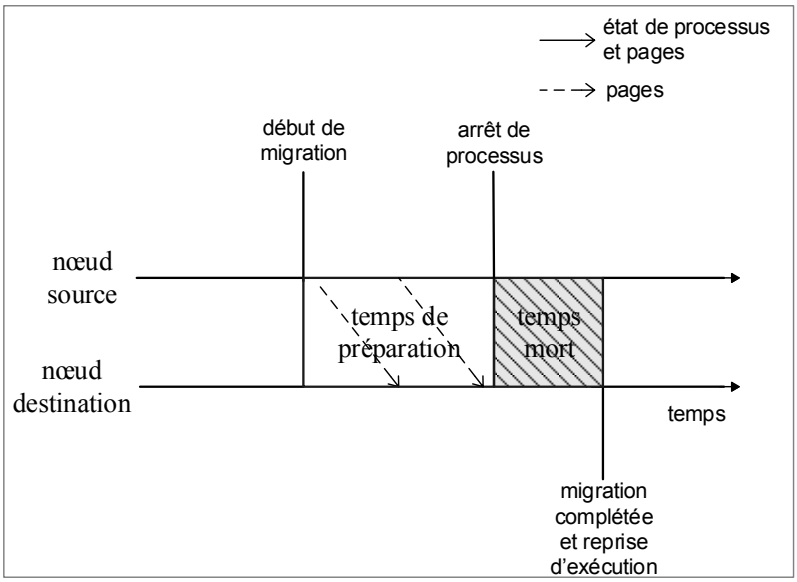


Figure 2.9 Périodes de transfert de l'algorithme de pré-copie (voir la définition à la page 28)

Un des inconvénients de cet algorithme est qu'il peut envoyer les mêmes pages plusieurs fois, ce qui augmente le nombre total de messages à transférer. Le fait qu'un processus peut

changer d'état pendant que son espace d'adressage est transféré demeure aussi un inconvénient. Un processus peut terminer de s'exécuter, par exemple. La figure 2.10 présente l'algorithme de pré-copie.

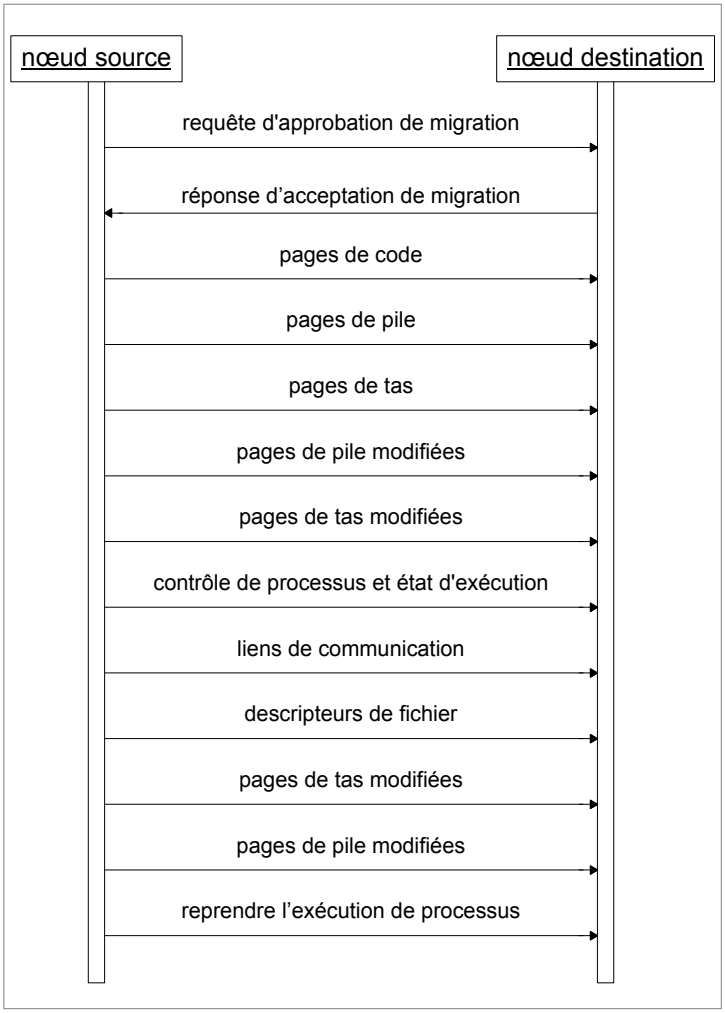


Figure 2.10 Algorithme de pré-copie

Algorithme de page à la demande

Zayas [Zayas, 1987a] a mis en œuvre l'algorithme de migration de processus de page à la demande. La figure 2.11 présente l'algorithme de page à la demande. Celui-ci utilise le mécanisme des pages sur référence où seules les pages référencées par le processus sont transférées. Il est semblable à l'algorithme de copie totale à la seule différence qu'aucune page

de mémoire virtuelle n'est transférée pendant la migration. En effet, seul l'état de processus et les descripteurs de fichiers sont transférés.

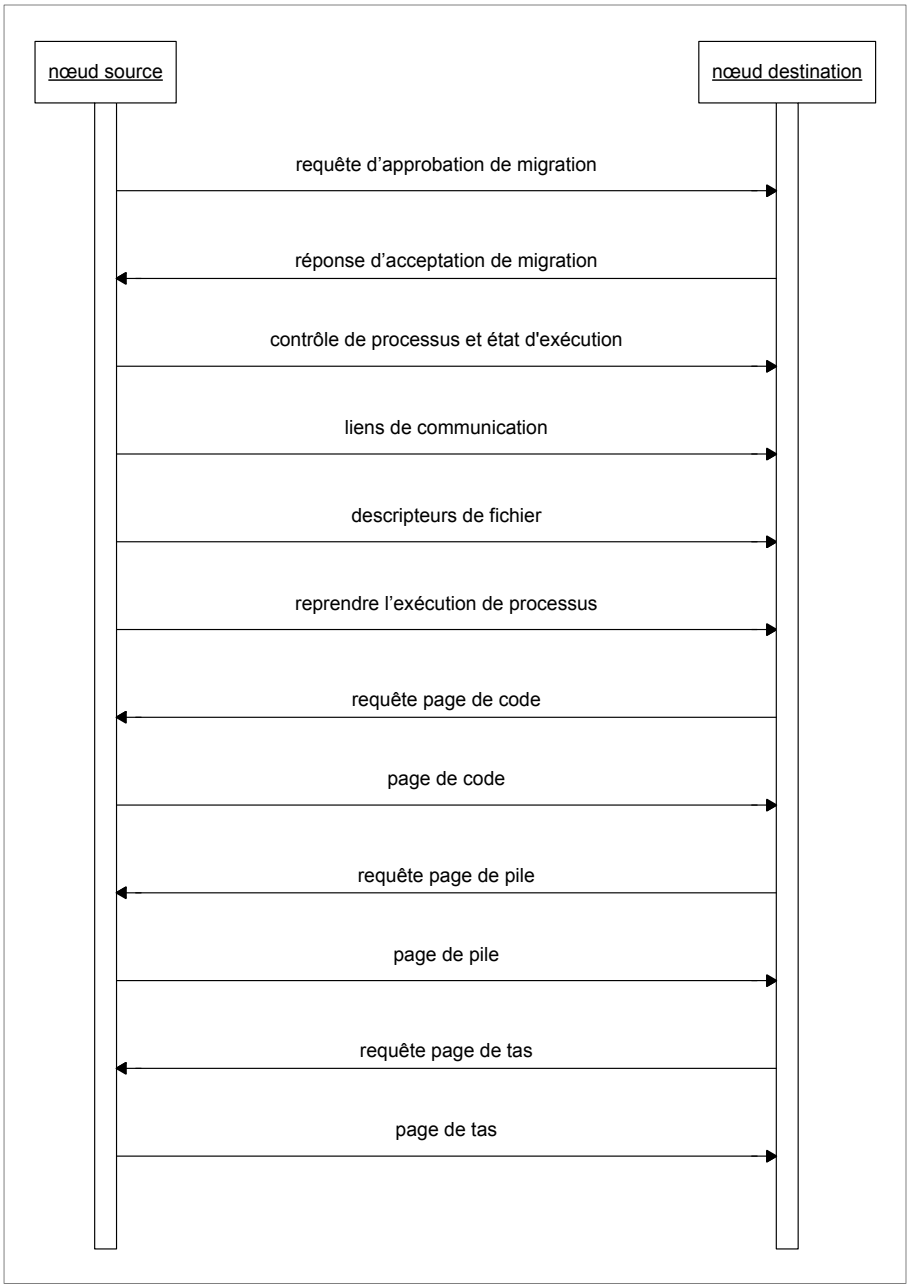


Figure 2.11 Algorithme de page à la demande

Une fois que le processus est redémarré dans le nœud destination, il envoie une requête de pages de code, de pile et de tas respectivement à l'hôte source. Les défauts de pages futurs

sont chargés au besoin. Cette méthode réduit le transfert de données de mémoire virtuelle et les pages sont envoyées au besoin. Il en résulte un gel de message plus court, une réduction du temps de migration de processus et une élimination du transfert de pages inutiles. L'inconvénient de cette méthode est que l'hôte source doit maintenir l'espace d'adresse virtuelle jusqu'à ce que le processus termine son exécution. En laissant l'espace d'adressage sur le nœud source, une dépendance résiduelle s'établit et ceci affecte la tolérance aux pannes du système. Ainsi, l'algorithme de page à la demande réduit le temps de transfert de processus mais entraîne un problème résiduel qui l'empêche d'être tolérant aux pannes. Comme le montre la figure 2.12, l'algorithme de page à la demande utilise les périodes de temps mort et de post-traitement pour la migration de processus.

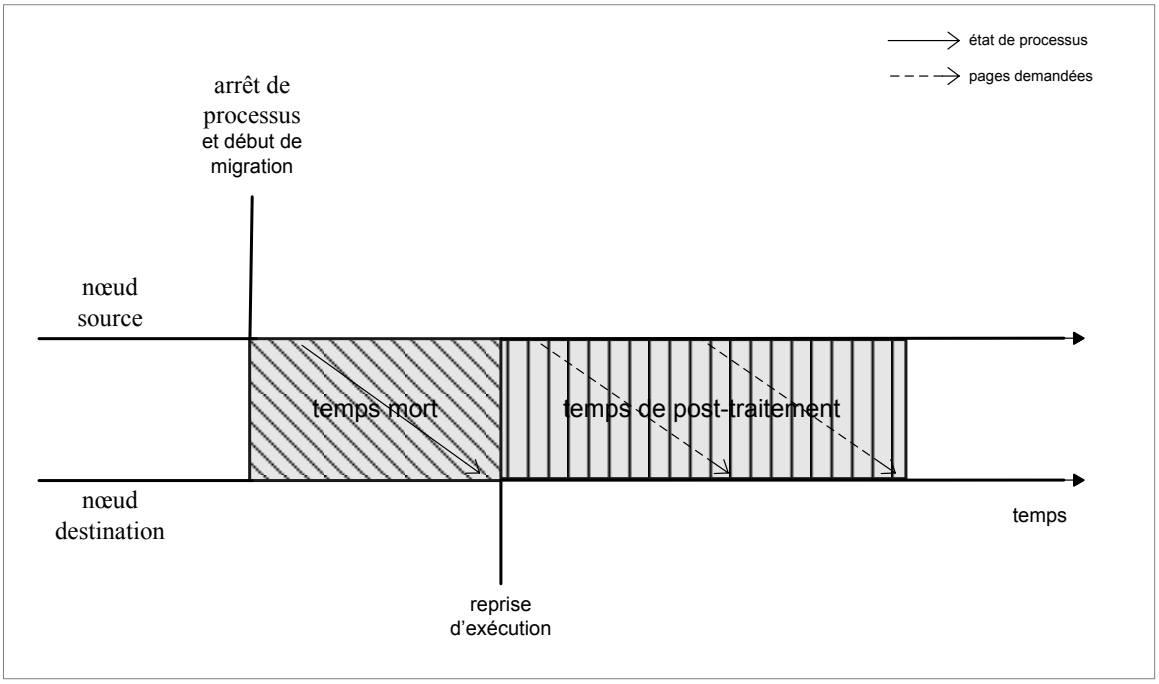


Figure 2.12 Périodes de transfert de l'algorithme de page à la demande

Algorithme de serveur de fichiers

L'algorithme de serveur de fichiers est très semblable à celui de page à la demande mais ajoute une troisième machine [Douglis, 1989] [Ousterhout et al., 1988]. Après la réception de la permission de migration de processus, le nœud source envoie l'état d'exécution,

l'information de lien de communication, les descripteurs de fichier et l'espace d'adressage. L'hôte source demande alors au destinataire de commencer à exécuter le processus. Ainsi au lieu de maintenir l'espace d'adressage au nœud source comme dans l'algorithme de page à la demande, il transfère simultanément les pages modifiées et les blocs de mémoire cache de fichier au serveur de fichiers. Selon les techniques précédemment utilisées, le nœud source devrait satisfaire les requêtes de page, mais avec cet algorithme, le serveur de fichiers manipulera les fautes de page produites par le processus migré une fois l'espace d'adressage transféré. La figure 2.13 présente les périodes de transfert de l'algorithme de serveur de fichiers.

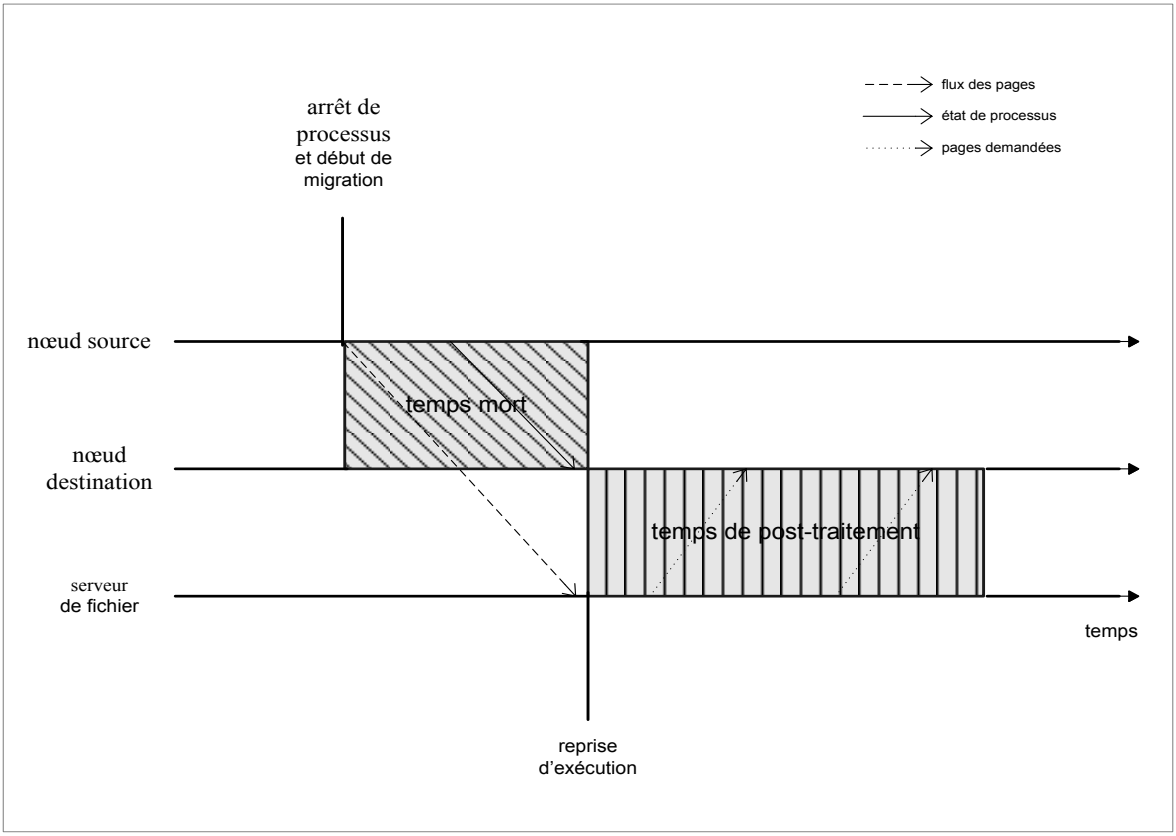


Figure 2.13 Périodes de transfert de l'algorithme de serveur de fichiers

Dans les algorithmes précédemment décrits, seules les machines source et destination ont été impliquées à la migration de processus. En ajoutant un serveur de fichiers, la migration de

processus peut utiliser la technique de page à la demande sans avoir une dépendance résiduelle à l'hôte source. La figure 2.14 l'algorithme de serveur de fichiers.

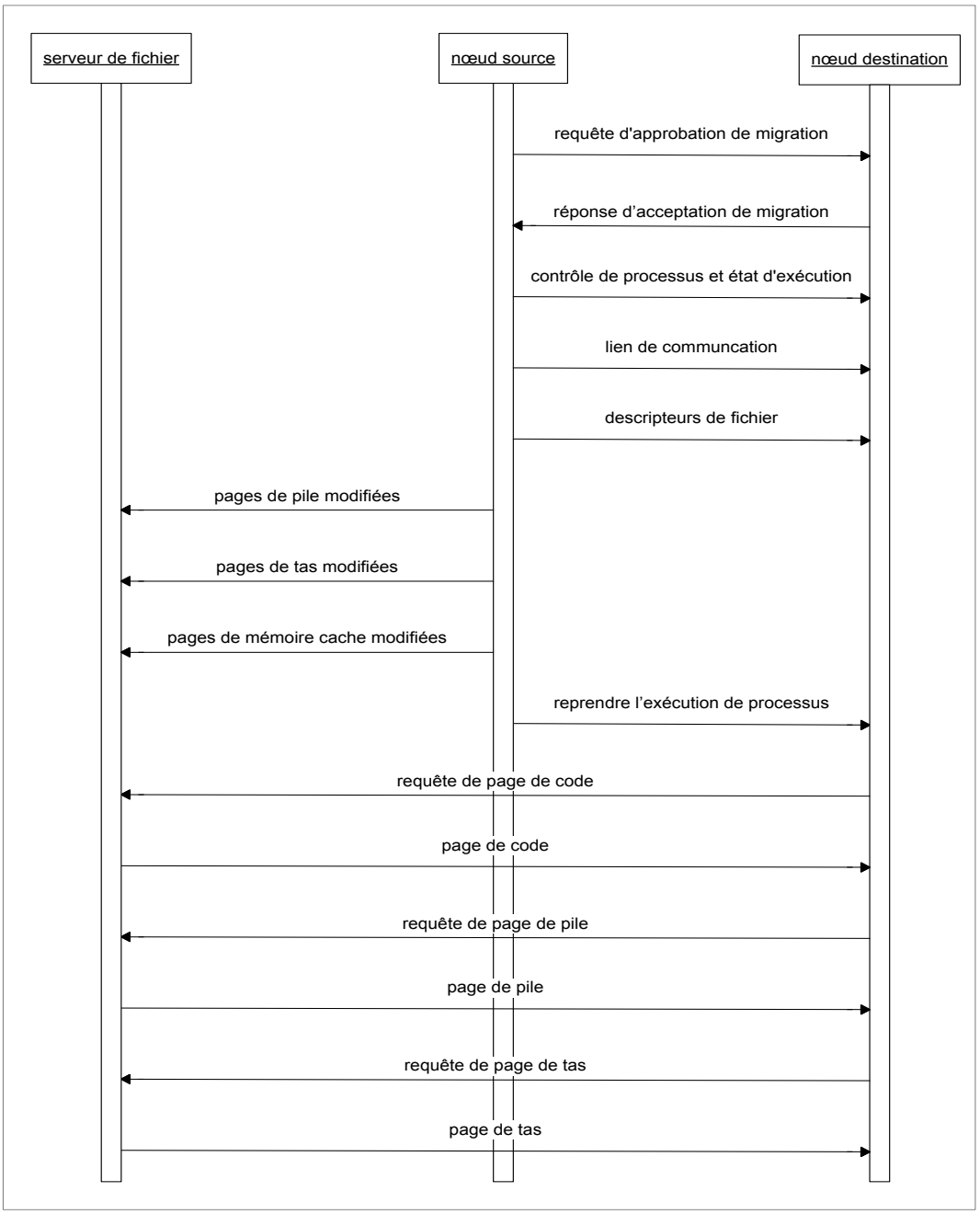


Figure 2.14 Algorithme de serveur de fichiers

Algorithme de gel libre

L'algorithme de gel libre a été proposé par Roush dans sa thèse de doctorat [Roush, 1995]. La motivation principale de cet algorithme est de réduire le temps de migration de processus qui, en moyenne, est de 50 millisecondes (ms) dans les algorithmes précédents. Cet algorithme peut migrer un processus en 14 ms. La réalisation de ce meilleur rendement sur les algorithmes de migration de processus précédents s'est faite en respectant les exigences suivantes. Premièrement, cet algorithme élimine la requête d'approbation en supposant que le nœud destination accepte la migration de processus et envoie le compteur ordinal et l'état d'exécution. Ceci fait économiser le temps de message va-et-vient qui est coûteux. Bien sûr, l'algorithme doit toujours redémarrer le processus de transfert si le nœud source rejette la migration. La figure 2.15 présente l'algorithme de gel libre.

L'algorithme de gel libre est destiné à un système distribué utilisant la communication de message. Il est supporté par un serveur de fichiers. Les principales étapes de l'algorithme sont:

- L'hôte source suspend le processus de migration;
- L'hôte source rassemble l'état d'exécution et le contrôle de processus et le transmet au destinataire;
- Le destinataire répond par un message d'acceptation ou de rejet;
- Simultanément après la transmission initiale, l'hôte source détermine la page de code, de pile et de tas puis les transmet;
- L'hôte destination reprend l'exécution de processus dès qu'il reçoit le premier page de code, de pile et de tas (facultative);
- Simultanément, l'hôte source transfère les restes de pages de pile, les liens de communication et les descripteurs de fichier à l'hôte destination;
- L'hôte source envoie simultanément les pages de tas et de mémoire cache modifiées au serveur de fichiers;
- L'hôte source envoie à l'hôte destination un flux de message complet indiquant que les pages sont disponibles au serveur de fichiers.

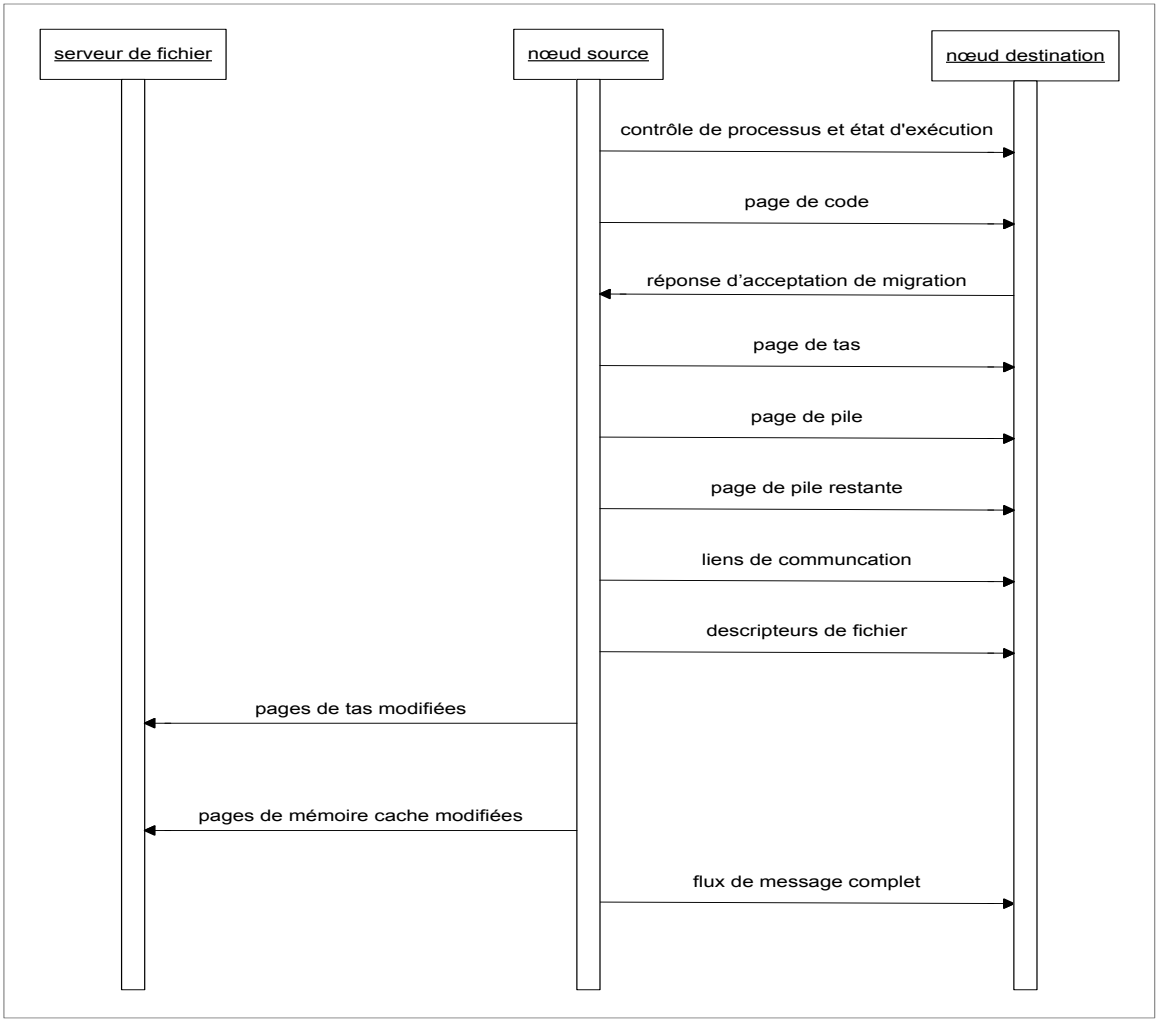


Figure 2.15 Algorithme de gel libre

La latence de migration de processus pour l'algorithme est le temps de la suspension de processus jusqu'à ce que l'hôte destination reçoive la première page de pile. Ce temps est constant indépendamment de la taille de l'espace d'adressage, du nombre de fichiers ouverts et du nombre de pages modifiées. Les messages vers le processus migrant sont traités par l'hôte source jusqu'à ce que la liaison soit transférée à l'hôte destination.

Algorithme de pré-copie de file en attente

L'algorithme de pré-copie de file en attente est une version légèrement améliorée de la technique de pré-copie. Il réduit le nombre de transferts de page multiples en assurant que les

pages souvent utilisées sont transférées moins souvent que possible. Une brève description des étapes effectuées pendant le temps de préparation est la suivante:

1. Un espace W contient toutes les pages d'un processus alors que le processus continue son exécution;
2. Chaque page de W est retirée de l'espace W, marquée en lecture seulement dans la table des pages, et elle est mise en attente dans une file R;
3. Les étapes précédentes sont reprises pendant un certain intervalle de temps jusqu'à ce que la migration soit terminée;
4. Si le processus lève un défaut de page en raison d'un accès en écriture à une page marquée en lecture seulement, la page est retirée de R et il est insérée dans W et marquée en écriture;
5. Un processus d'arrière-plan retire continuellement la première page de R de la file d'attente et la transfère à l'hôte de destination;
6. Si R est vide, la taille de W peut être mesurée. Si la taille est inférieure à une limite fixée, le processus de migration est interrompu et toutes les pages de W sont mises dans la file d'attente de R et elles sont transférées.

Cet algorithme a été décrit dans *Nomadic Operating System* [Henriksen et Hansen, 2002] pour la première fois. Il a été implémenté dans Smile [Noack, 2003] sur Linux.

Algorithme de post-copie

L'algorithme de post-copie [Richmond, 1996] ressemble beaucoup aux algorithmes précédemment décrits. Initialement, cet algorithme transfère l'état de processus. Il évite les dépendances résiduelles en transférant l'information de processus restante dès que le processus reprend son exécution à l'hôte de destination. La migration de processus suit les étapes suivantes:

1. La suspension du processus de l'hôte source;
2. Le transfert de l'état de processus vers le nœud destination;
3. La reprise de l'exécution du processus migré;

4. Le transfert de toute l'information de processus restante tant que le processus continue son exécution sur l'hôte destination;
5. Le transfert des pages restantes du processus s'exécutant sur le nœud source.

Pendant l'exécution du processus dans l'hôte de destination, le transfert de pages doit être transparent. Comme l'algorithme de page à la demande, le processus peut faire référence aux pages de son espace d'adressage qui résident toujours sur l'hôte source. Pour pallier au problème de dépendances résiduelles, une requête prioritaire est envoyée à l'hôte source. Puis, la page demandée est envoyée immédiatement. Cependant, la requête prend un laps temps et l'exécution de processus est suspendue pendant cette période. Il prend moins de temps pour résoudre une faute de page localement que via un réseau. Les dépendances résiduelles existent seulement pendant la période de post-traitement où les pages restantes sont transférées de l'hôte source. Cet algorithme est implémenté dans Smile [Noack, 2003]. La figure 2.16 présente les périodes de transfert de l'algorithme de post-copie.

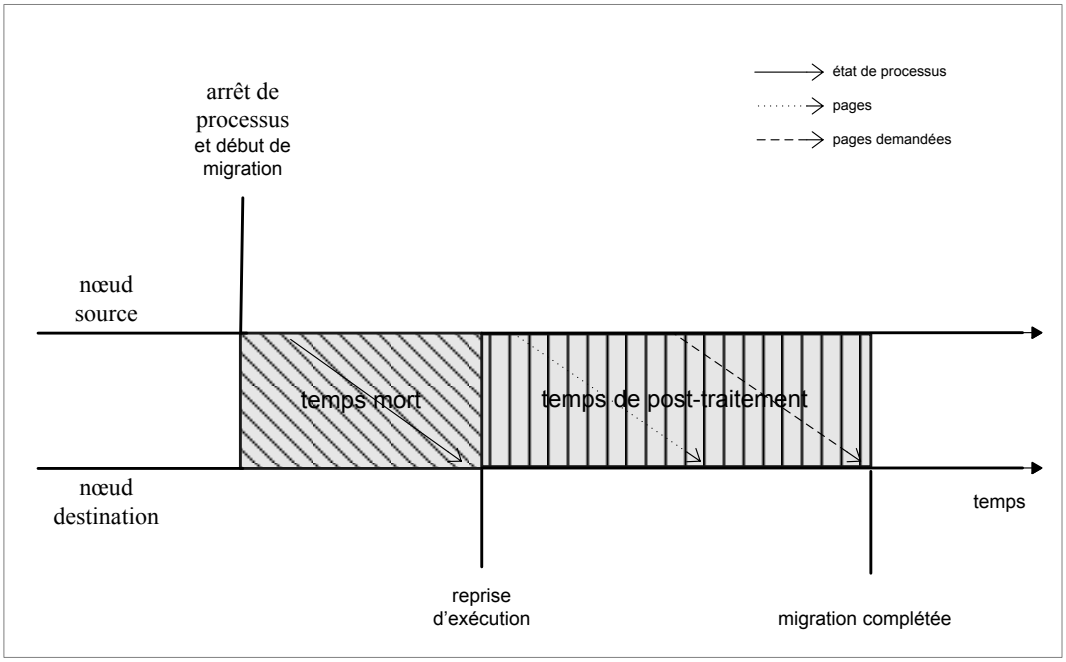


Figure 2.16 Périodes de transfert de l'algorithme de post-copie

Algorithme de post-copie assistée

La plupart des algorithmes précédemment décrits utilisent deux périodes pour la migration de processus. L'algorithme de post-copie assistée a été inventé pour utiliser toutes les trois périodes de migration de processus dans Smile [Noack, 2003].

L'algorithme de post-copie assistée est essentiellement une synthèse de pré-copie et de post-copie. C'est la première fois qu'un algorithme de migration de processus utilise toutes les trois périodes pour la migration de processus (voir la figure 2.17). Brièvement, cet algorithme exécute les étapes suivantes:

1. Le transfert de l'espace d'adressage entier pendant que le processus continue à s'exécuter sur l'hôte source;
2. Le stockage temporairement des pages reçues par l'hôte de destination;
3. La suspension du processus;
4. Le transfert de l'état de processus;
5. La reprise de l'exécution de processus au nœud destination;
6. Le transfert de toutes les pages modifiées du nœud source;
7. L'envoi immédiat des pages demandées par le processus en exécution;
8. Le stockage de toutes les pages non modifiées dans un espace d'adressage temporaire de l'hôte de destination.

Comme l'algorithme de pré-copie, l'espace d'adressage entier du processus d'exécution est initialement transféré pendant le temps de préparation. Les pages sont stockées dans un espace d'adressage temporaire de l'hôte de destination au lieu d'être insérées dans l'espace d'adressage du nouveau processus. Après le transfert initial, l'hôte source suspend le processus de migration et expédie l'état de processus. Le processus migré reprend son exécution immédiatement.

Quand le processus reprend son exécution au nœud destination, il commence à faire référence aux pages. Immédiatement une faute de page est générée en raison de l'espace d'adressage vide. Le fait de résoudre la faute de page en se référant aux pages stockées dans l'espace

d'adressage temporaire à l'hôte de destination peut provoquer l'incohérence parce qu'elle pourrait être modifiée pendant le temps de préparation à l'hôte source. Une requête est envoyée à l'hôte source. Dans le cas où la page a été modifiée, elle serait transférée de nouveau. Sinon, le processus en exécution peut se référer aux pages stockées dans l'espace d'adressage temporaire du nœud destination. Toutes les pages qui ne résident pas dans l'espace d'adressage du processus ont besoin de ce contrôle de cohérence.

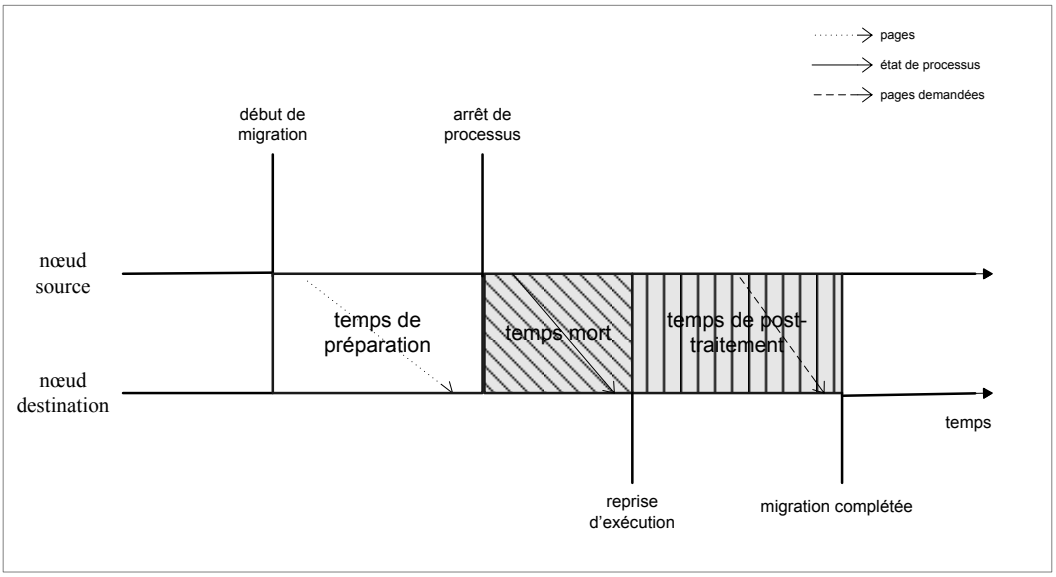


Figure 2.17 Périodes de transfert de l'algorithme de post-copie assistée

Pour éviter que chaque page soit vérifiée, l'hôte source transfère les pages modifiées pendant l'exécution du processus migré. Ces pages seront donc cohérentes et peuvent ainsi être insérées dans l'espace d'adressage de processus. Une fois toutes les pages modifiées se trouvent sur l'hôte de destination, le contrôle de cohérence n'est plus nécessaire. Toutefois, il arrive que quelques pages ne soient pas toujours dans l'espace d'adressage puisqu'elles sont restées inchangées sur l'hôte source. Ces pages doivent être expédiées au nœud destination pour être stockées dans l'espace d'adressage temporaire du processus.

L'amélioration principale de cette technique se situe au niveau de l'élimination de la limite prédéterminée nécessaire de *pré-copie* et de *pré-copie de file en attente*. Contrairement aux autres algorithmes, celui de *post-copie assistée* transfère l'espace d'adressage du processus une seule fois durant le temps de préparation. Comme dans l'algorithme de post-copie, les

dépendances résiduelles apparaissent seulement pendant la période du transfert de pages modifiées de l'hôte source.

La plupart des algorithmes existants transfèrent chaque page une seule fois mais dans *pré-copie* et *pré-copie de file en attente*, il est ainsi difficile de déterminer d'avance le nombre de fois qu'une page sera transférée. Cela dépend du nombre de fois que le processus a accédé à la page.

Grâce à l'algorithme de *post-copie assistée*, il est maintenant possible de déterminer le nombre de transferts de pages multiples. Chaque page est transférée deux fois au maximum. Premièrement, durant l'espace d'adressage initial et deuxièmement, pendant le temps de préparation. Le nombre de transferts de page multiples s'en trouve ainsi réduit.

2.3.2 Plateformes de migration de processus

MOSIX

MOSIX est un système d'exploitation distribué développé par une équipe de l'université hébraïque de Jérusalem. Ce système est destiné à l'origine pour le monde Unix et les ordinateurs spécialisés (comme les VAX 780). En 1992, MOSIX s'oriente vers le monde du PC. Depuis cette date, ce système est passé d'une plateforme de développement UNIX BSD à une plateforme Linux. Le système MOSIX permet de créer une grappe (cluster) de PC qui se comporte quasiment comme une machine multiprocesseurs. Les algorithmes de MOSIX travaillent au niveau du noyau et permettent de faire collaborer les machines ensemble en échangeant des processus et en répartissant la charge dans le cluster.

MOSIX utilise l'algorithme de page à la demande. L'architecture de son noyau se divise en trois couches [Milojicic et al., 2000]: supérieure, liaison et inférieure. Chaque objet dans MOSIX a un indicateur universel unique. Le noyau peut faire référence à n'importe quel objet dans la grappe. La couche supérieure fournit l'interface du système sur chaque nœud et manipule seulement des objets universels. Le noyau inférieur fournit des services tels que les

gestionnaires de périphérique et le changement de contexte. La troisième, la couche liaison, fournit la communication entre les nœuds, le transfert de données, la migration de processus et les algorithmes de répartition de charges. En effet, chaque processus peut être migré entre les nœuds afin de tirer avantage de la meilleure ressource disponible. L'idée principale consiste à répartir sur plusieurs machines, non pas le calcul, mais le multitâche. Quand le noyau supérieur doit exécuter une opération sur un objet universel, il utilise la couche liaison.

Dans un cluster MOSIX, le processus est le plus petit élément gérable. La coopération entre les nœuds du cluster se fait au niveau du processus. Le principe de base de MOSIX consiste à faire migrer un processus de façon dynamique et préemptive vers le nœud du cluster qui présente les meilleures performances. La recherche automatique d'une performance optimale pour l'exécution des processus est un des principaux avantages de MOSIX. En effet, lorsque des processus sont créés, MOSIX calcule et détermine automatiquement quel est le nœud le moins chargé en terme d'utilisation du processeur et d'occupation de la mémoire RAM, et fait migrer une partie du processus en cours d'exécution vers la meilleure machine: c'est à dire celle qui offre le processeur le plus rapide, le moins occupé et la mémoire la moins saturée.

Charlotte

Le système de migration de processus Charlotte [Finkel et Artsy, 1989] utilise l'algorithme de copie totale. Les processus communiquent entre eux via des liens. Dans Charlotte, la migration se divise en trois phases:

- La négociation: le noyau du nœud source envoie une demande de migration au destinataire. Si la demande est acceptée, le nœud source envoie au destinataire la taille du processus, la table de liens, les informations sur le processus et l'utilisation du réseau. Si le destinataire a les ressources nécessaires demandées par le nouveau processus, il l'indique au mécanisme d'initialisation de la migration. Ce dernier réserve alors toutes les ressources auprès du noyau en il lui répondant par un message d'acceptation. Puis, c'est le début de migration de processus.
- Le transfert: Durant cette phase, le processus migrant est gelé. Le transfert est réalisé en trois phases:

  - Le transfert de l'image du processus vers le destinataire où le processus est chargé dans l'espace réservé par le noyau durant la phase de négociation;
  - La mise à jour des liens durant laquelle le noyau de chaque processus migrant modifie les nouvelles adresses des liens;
  - Le transfert de l'environnement où toutes les structures de données du noyau sont transférées (descripteurs de processus, de liens, de messages reçus et en attente).
- La libération de ressources: après l'envoi de tous les paquets, le noyau source supprime toutes les structures de données appartenant au processus migré.

Système V

Le système V [Theimer et al., 1985] est basé sur l'échange de messages entre machines. Un mécanisme d'exécution à distance permet de s'adresser à un groupe de processus et un autre mécanisme assure la migration. L'algorithme pré-copie permet au processus d'être copié sur le nœud destinataire avant le gel du processus. Cette anticipation permet de gagner du temps. Ce système utilise la notion de processeur hôte logique, qui est défini par un espace d'adressage dans lequel plusieurs processus peuvent s'exécuter. La migration se fait en continuant l'exécution sur la machine source. Seules les pages modifiées seront transmises à la fin du transfert. Le contexte est transféré au nœud cible pendant que le processus continue de s'exécuter. Nous avons donc deux processus: un sur le nœud source et l'autre sur le destinataire.

La détermination de l'adresse destinataire est réalisée par communication interprocessus et un mécanisme de migration qui suit les étapes suivantes:

- L'initialisation de l'hôte destinataire. À l'aide des descripteurs envoyés par l'hôte source, le noyau du destinataire va créer une nouvelle copie du processus situé sur la machine source.
- La pré-copie de l'environnement du processus migrant vers le nouveau nœud.

- La terminaison de la copie où le processus destinataire est gelé et la copie de son environnement complétée. Toutefois, les messages qui arrivent doivent être stockés pour ensuite être traités.
- Le réveil de la nouvelle copie et rétablissement des liens. L'ancien processus est effacé sur la machine source et le nouveau est activé. L'identificateur de processus est lié au nouveau processus qui est en liaison physique avec la machine destinataire grâce à un cache situé sur chaque noyau.

Accent

Ce système utilise l'algorithme de la page à la demande pour effectuer la migration. Ainsi, on évite le transfert effectif des données. Le transfert réel a lieu lorsque le destinataire utilise explicitement les données. Le transfert se fait page par page selon la demande. Ainsi, on diminue le temps de transfert. Au début de l'opération de migration, seul le contexte est transféré.

Le travail dans [Rashid et Robertson, 1981] compare avec précision la méthode de copie totale et celle de la page à la demande. Les résultats publiés montrent que le nombre d'octets transférés diminue de 58 % en moyenne. Cependant les deux machines (source et destination) restent liées pendant toute la migration.

Sprite

Sprite [Douglis et Ousterhout, 1991] est un système d'exploitation à partage de charge implémenté par University of California, Berkeley, au début des années 1990. Ce système met en œuvre la migration préemptive. Certains des processus, choisis selon des critères bien particuliers, migrent vers des machines inutilisées du réseau. Quand la machine distante sur laquelle ces processus sont exécutés est à nouveau utilisée, ils sont immédiatement rapatriés. Ils peuvent alors migrer vers un autre hôte inactif mais doivent dans tous les cas repasser par leur machine d'origine.

Sprite est le premier système qui a surmonté le problème de dépendances résiduelles de l'hôte source en employant l'algorithme de serveur de fichiers. Lors de la migration de processus, son exécution est interrompue de l'hôte source et toute la mémoire de son espace d'adressage est ainsi transférée au serveur de fichiers. L'état de processus est transféré comme d'habitude et chaque page peut être demandée au serveur de fichiers. Le système Sprite assure que, dans la plupart des cas, la page ne provoque pas d'opérations de disque parce que le serveur de fichiers utilise sa mémoire comme une antémémoire.

Choices

Le système Choices [Campbell et al., 1993] utilise l'algorithme de gel libre pour la migration de processus. Le travail dans [Roush, 1995] décrit la conception et la mise en œuvre de cette technique. En outre, des expériences de normalisations de la performance d'algorithmes de migration de processus ont été faites dans ce travail pour permettre une évaluation de différentes stratégies de migration.

L'algorithme de serveur de fichiers mis en œuvre dans Sprite et l'algorithme de page à la demande implémenté dans l'Accent sont comparés à l'algorithme de gel libre. Les résultats montrent des avantages de l'algorithme de gel libre par rapport aux deux autres algorithmes. Cependant, la normalisation de la performance d'algorithmes de migration de processus est souvent difficile à obtenir [Noack, 2003].

RHODOS

RHODOS [Goscinski, 1994] est un système d'exploitation distribué expérimental développé par Deakin University en Australie. Il consiste en micronoyau et fournit la transmission de message. RHODOS a été conçu pour utiliser différentes stratégies de migration et de les comparer. Il permet de déterminer quel algorithme de migration peut être utilisé pour réaliser la meilleure performance et dans quelles conditions.

Une comparaison de performance entre un algorithme de page à la demande (la stratégie de copie-sur-référence) et un algorithme de copie totale sur RHODOS ont été publiée dans [De paoli, 1994]. La seule métrique mesurée et évaluée était le temps mort.

Avec l'algorithme de la copie totale, le temps mort était trois fois plus élevé que celui de la page à la demande sur le même processus [Noack, 2003]. Cependant, des métriques détaillées des différentes périodes utilisées par la migration de processus n'ont pas été décrites.

Mach

Le micronoyau Mach [Milojicic et al., 1992] a été développé par Carnegie Mellon University, aux États-Unis. Le mécanisme de migration du système Mach a été inventé par University of Kaiserslautern en Allemagne.

Mach met en œuvre deux sortes de mécanismes qui fournissent différents algorithmes de migration de processus. Le serveur de migration simple (en anglais *Simple Migration Server*), très robuste, a été implanté en premier. Il met seulement en œuvre la page à la demande pour la migration de processus. Le serveur de migration optimisé (en anglais *Optimized Migration Server*) fournit, quant à lui, la migration de processus au niveau application et supporte la page à la demande, la copie totale et l'algorithme de pré-copie.

Smile

Smile est un système d'exploitation pour la migration de processus développé par Dresden University of Technology en Allemagne. Ce système a été implémenté sur Linux et a les caractéristiques suivantes [Noack, 2003]:

- La migration de service est complètement transparente à l'utilisateur et les applications n'ont pas besoin d'être recompilées.
- La migration n'a pas de dépendances résiduelles: Smile supporte la migration de service complète sans laisser des ressources sur la machine source. Cependant, durant le temps de post-traitement les dépendances résiduelles sont tolérées pour le transfert de données.

- Il résulte en fait d'une modification minimale du noyau Linux. Pour la migration de processus, ce système nécessite une modification et peut-être une création de nouvelles instances de structures de données clé du système d'exploitation. La plupart de ces structures sont seulement accessibles au niveau noyau comme l'information d'espace d'adressage (la table de page). Une certaine forme de support pour la migration de processus d'un système d'exploitation est donc exigée. Notez que Linux ne supporte pas la migration de processus sans la modification du noyau.

Le système Smile implémente le service de migration au niveau du noyau sur Linux. Ainsi, l'assistance du noyau est indispensable. Une modification du noyau est toujours une approche simple mais sa nouvelle fonction est inflexible [Noack, 2003]. Une alternative à cette modification est un module de noyau dynamiquement chargeable. L'avantage de cette approche réside sur le fait qu'aucune recompilation du noyau n'est nécessaire et le support de migration peut être ajouté ou enlevé du noyau lors de l'exécution. Le module utilise une interface bien définie pour accéder aux structures et fonctions du noyau. Ainsi, l'ajout dynamique de nouvelles fonctions est plus approprié que la modification du noyau Linux.

Le système Smile n'est pas limité à un certain algorithme de migration de processus et fournit une interface bien définie pour intégrer les autres techniques de migration. Grâce à ces fonctions, Smile fournit un mécanisme de migration qui est approprié pour la mise en œuvre et l'évaluation d'algorithmes de migration de processus.

2.3.3 Évaluation des algorithmes de migrations de processus

Dans cette section, nous allons présenter les évaluations des algorithmes précédemment décrits.

Évaluation Qualitative

Dans l'algorithme de *copie totale*, le *temps mort* est égal à la durée de la migration totale. D'une part, on s'attend à ce que le *temps mort* de l'algorithme *de copie totale* soit le plus élevé de tous les algorithmes car toutes les informations de processus sont transférées au cours de

cette période. D'autre part, il est prévu que le temps de la migration totale de l'algorithme de *copie totale* soit le plus bas parce que ses informations doivent être transférées seulement une fois. Par ailleurs, l'algorithme *de copie totale* n'utilise ni *temps de préparation*, ni *temps de post-traitement*.

Contrairement à l'algorithme de *copie totale*, celui de *pré-copie* exploite le *temps de préparation* pour transférer les informations avant le départ réel du processus. Ceci réduit la quantité de données transférées pendant le *temps mort*. Par conséquent, avec l'algorithme de *pré-copie*, le *temps mort* est beaucoup plus court par rapport à celui de *copie totale*, mais certainement plus élevé que celui de *post-copie* ou celui de *post-copie assistée*. Cela dépend du nombre de pages qui doivent être transférées.

Une comparaison de la durée de la migration totale est difficile à obtenir car elle est fortement affectée par le *temps de préparation* et le nombre de transferts de page multiples. Si le processus de migration modifie un grand nombre de pages, on s'attend à ce que le temps de migration totale de l'algorithme de *pré-copie* soit le plus élevé de tous les algorithmes de migration. Par rapport à l'algorithme de *pré-copie*, celui de *pré-copie de file en attente* réduit le nombre de transferts de page multiples et donc le temps de migration totale. Par ailleurs, on s'attend à ce que le *temps mort* de l'algorithme de *pré-copie de file en attente* soit légèrement inférieur au *temps mort* de celui *pré-copie*.

Il est prévu que l'algorithme de *post-copie* ait un *temps mort* significativement plus court parce qu'il envoie seulement un minimum d'état de processus durant cette période. Pendant le *temps de post-traitement*, le processus migré demande les pages à l'hôte source. Ainsi, l'exécution du processus migré est significativement retardée. Il en résulte qu'un *temps de migration totale* plus long comparé à l'algorithme de *copie-totale*.

L'algorithme de *post-copie assistée* donnera probablement le *temps mort* le plus court parce qu'il transfère un état du processus plus réduit à l'hôte de destination. L'algorithme utilise à la fois le *temps de préparation* et le *post-traitement* mais chaque période est plus courte par

rapport aux algorithmes *pré-copie et post-copie*. En outre, on s'attend aussi que l'algorithme de *post-copie assistée* aboutisse à un temps de migration totale le plus élevé.

Cette évaluation qualitative présente des hypothèses. Pour les confirmer, il faut mesurer les différentes périodes dans un système qui implémente ces algorithmes.

Évaluation Quantitative

Le tableau 2.1 présente les caractéristiques des algorithmes précédemment décrits. Pour chaque algorithme, ce tableau indique si l'algorithme utilise le *temps de préparation*, le *temps mort (suspendu)* et le *post-traitement*. De plus, il présente les dépendances résiduelles et les systèmes où l'algorithme particulier a été implémenté.

Tableau 2.1 Caractéristiques des algorithmes de migration de processus

Période / Algorithme	Temps de préparation	Temps mort	Temps de post-traitement	Dépendances résiduelles	Systèmes
Copie totale	non	oui	non	non	RHODOS Charlotte Mach
Page à la demande	non	oui	oui	oui, de l'hôte source	RHODOS MOSIX Mach Accent
Serveur de fichiers	non	oui	oui	oui, du serveur de fichiers	Sprite
Pré-copie	oui	oui	non	non	Système V Mach
Pré-copie de file en attente	oui	oui	non	non	Linux
Post-copie	non	oui	oui	non	Linux
Gel libre	non	oui	oui	oui, du serveur de fichiers	Choices
Post-copie assistée	oui	oui	oui	non	Linux

Le tableau 2.1 montre que la plupart des algorithmes exploitent deux périodes et certains d'entre eux ont besoin d'une troisième entité pour éviter les dépendances résiduelles de l'hôte source. La plupart des algorithmes ont été mis en œuvre sur différents systèmes, par conséquent une comparaison quantitative directe est difficile à obtenir.

Lors de la migration, la suspension de l'exécution du processus et l'interruption de communication avec le monde extérieur joue un rôle essentiel. Le *temps mort* d'un processus est donc la période la plus importante pour évaluer la performance d'un algorithme. L'autre métrique importante est le temps de migration totale. En plus des périodes, le retard d'exécution reste un élément déterminant sur la performance de la migration de processus.

Un certain nombre de travaux publiés sur la performance du processus de migration ont mesuré le *temps mort* d'un processus de référence de 100 kilooctets [Noack, 2003] [Roush, 1995] [De paoli, 1994] [Finkel et Artsy, 1989] [Steketee et al., 1994]. Le tableau 2.2 présente l'année, l'environnement de test et les algorithmes employés. Le tableau 2.3 présente les résultats de processus de référence avec une bande passante de 10 Mo/s sur l'environnement Smile [Noack, 2003].

Tableau 2.2 Systèmes de migration de processus et le temps mort

Système d'exploitation	Année	Plateforme	Réseau Mb/s	Temps mort (ms)	Algorithme
Système V	1985	Sun 10MHz68010	Ethernet 10	680	Pré-Copie
Accent	1987	Perq workstation	Ethernet 10	13 000	Page à la demande
Charlotte	1989	VAX 11/750	Pronet	750	Copie totale
Sprite	1991	SPARCstation1	Ethernet 10	330	Serveur de fichiers
Mach OMS	1993	Intel80486 33MHz	Ethernet 10	250	Page à la demande
Choices	1995	SPARCstation2	Ethernet 10	14	Gel libre
RHODOS	1997	SUN 3/50	Ethernet 10	118 351	Page à la demande Copie totale
Linux	2003	Dual Pentium 450 / Dual Athlon 1800	Ethernet 10	89 33 5	Copie totale Post-copie assistée Post-copie

Tableau 2.3 Périodes du processus de référence utilisant 10 Mo/s

Période / Algorithme	Temps de préparation (ms)	Temps mort (ms)	Temps de post-traitement (ms)	Temps total (ms)
Copie totale	0	88.5	0	88.5
Pré-copie	52.4	36.5	0	88.9
Pré-copie en file d'attente	37.1	51.4	0	88.5
Post-copie	0	4.9	93.4	98.3
Post-copie assistée	52.4	33.4	3.5	89.3

Les algorithmes de transfert d'un processus se distinguent, d'une part, selon la répartition de l'opération de transfert dans le temps, et d'autre part, selon les ressources mises en œuvre pour

le transfert. Il est donc important de réduire le *temps mort* sans toutefois augmenter le *temps total* de transfert.

2.4 Migration de processus dans des systèmes hétérogènes

Dans cette section, nous proposons une synthèse de certains nombres travaux sur la migration de processus dans des environnements hétérogènes et leur évaluation. Toutefois, la définition du concept de migration de processus dans des systèmes hétérogènes et la spécification des critères de performances s'impose dans un premier temps.

2.4.1 Concept de migration de processus dans des systèmes hétérogènes

La migration de processus dans des systèmes hétérogènes peut être définie comme la capacité à déplacer un processus en cours d'exécution entre les différents processeurs qui sont reliées uniquement par un réseau (c'est-à-dire n'utilisant pas une mémoire localement partagée). Le système d'exploitation de la machine d'origine doit emballer tout l'état du processus de sorte que la machine de destination peut continuer son exécution. Le processus ne devrait normalement pas être concerné par les changements de son environnement autre qu'une variation de performance. En effet, lors de déplacement des pages de mémoire ou la capture et la restauration de l'état du processus, les liaisons de communication du processus doivent être maintenues. Une conception soignée du mécanisme d'un système d'exploitation de communication interprocessus peuvent faciliter la migration d'un processus.

Dans les systèmes homogènes, la migration de processus est basée sur le fait que les machines source et destinataire ont la même architecture. C'est-à-dire leurs unités centrales de traitement comprennent le même ensemble d'instruction et leurs systèmes d'exploitation ont des ensembles d'appels de système et de conventions de mémoire similaires. Ceci permet aux informations d'état d'être copiées mot à mot entre les hôtes pour que l'image de la mémoire soit identique.

La migration de processus dans les systèmes hétérogènes élimine cette supposition des machines sources et destinataires de même architecture. En plus des questions de migration sur les machines homogènes, il faut des mécanismes de traduction de l'état complet du processus de sorte qu'il peut être compris par la machine de destination. Cela nécessite la connaissance du type et de l'emplacement de toutes les valeurs de données (dans des variables globales, les trames de pile et le tas).

Avant d'aborder les différentes études dans le domaine, il est nécessaire d'examiner les diverses classes de migration hétérogène ou des systèmes de mobilité qui existent déjà. Notre discussion porte sur l'unité d'information en cours de migration et décrit comment cette information peut être déplacée. Les migrations de processus dans des systèmes hétérogènes peuvent être classifiées en:

- *Migration d'un processus avec un code interprété*: Lorsqu'un processus courant s'exécute en utilisant un interprète, la migration du processus implique la traduction de l'état de l'interprète et toutes les valeurs de données auxquelles il peut accéder. Si ces valeurs (c'est-à-dire les variables, les paramètres et les autres données sur la pile d'appel) sont stockées de façon indépendante de la machine, la migration est simple.
- *Migration d'un processus avec un code natif*: Si le programme actif est compilé en code machine natif, la sauvegarde l'état est plus difficile. En effet, chaque machine possède sa propre méthode de stockage des valeurs du programme. La disposition de pile, l'utilisation de registres et la structure du code exécutable sont très différentes entre les machines.

Les autres aspects qui devraient être considérés lors de la conception d'un système de migration sur les machines hétérogènes comprennent:

- Une *migration auto-initiée*: Le processus en cours de migration décide quand et vers où migrer plutôt que laisser ce choix à un agent externe comme le système d'exploitation. Ceci assure que le processus est dans un état bien connu (par exemple un appel d'une procédure permettant la migration) plutôt qu'un point relativement aléatoire dans le code. La migration dans un état connu réduit la quantité d'informations à être migrée et simplifie sa reconstruction.

- Un *code supplémentaire dans le processus de migration*: L'ajout de code (explicitement par le programmeur ou implicitement par le compilateur) pour la capture et la restauration des données de processus peut simplifier la migration. Ceci peut se présenter sous la forme d'une bibliothèque spéciale qui doit être liée au programme ou sous la forme d'un code supplémentaire au début et à la fin de chaque procédure mais présente l'inconvénient d'induire un temps d'exécution supplémentaire.
- Un *type-sécurité*: Dans un langage de type sécurisé, chaque variable dans un programme a un type. Le langage garantit que les valeurs stockées dans des variables soient toujours du type approprié. Les variables sont toujours initialisées à la valeur par défaut de leur type avant que toute référence à la variable ne puisse être effectuée. La mémoire non initialisée est protégée. Il est donc impossible d'accéder à un espace mémoire non initialisé. Les systèmes de migration de processus dans des machines hétérogènes exigent que le processus migrant doive être écrit, soit dans un langage totalement de type sécurisé, soit dans un sous-ensemble de type sécurisé. L'algorithme de migration exige l'information complète de type, d'habitude produites par un compilateur, les données de la machine de destination. Si les données de type sont incohérentes en raison des manques dans le langage de mise en œuvre, la migration devient plus difficile ou même impossible.

2.4.2 Critères de performances

Les performances d'un mécanisme de mobilité sont évaluées selon trois critères: la latence de la mobilité, le surcoût sur l'application mobile et le surcoût sur les applications non mobiles. Notez qu'à la suite de cette section, nous allons utiliser le mot processus, agent et application de manière indifférente pour alléger la description.

La latence de la mobilité d'une application est le temps nécessaire pour déplacer l'application d'un site source vers un site destination. Ce temps est borné par l'instant d'appel d'une primitive de la mobilité sur l'application et l'instant d'arrivée de l'application mobile sur son site destination avant le lancement de son exécution. La latence de la mobilité dépend du temps d'accès et de la manipulation des structures internes de l'application (état des données,

état de l'exécution), d'une part, et de la taille de l'état de l'application, c'est à dire de l'état qui sera transféré sur le réseau, d'autre part.

Une remarque s'impose toutefois: le niveau de mise en œuvre du mécanisme de la mobilité est un paramètre qui peut avoir un rôle déterminant sur la latence. Il peut être réalisé au niveau de l'application (niveau "middleware") ou au niveau du système d'exploitation. Généralement, lorsque le niveau de mise en œuvre de la mobilité est bas (niveau de système d'exploitation), les informations sur l'état d'exécution de l'application sont directement accessibles et utilisables. C'est à ce niveau que l'accès et la manipulation de l'état d'exécution devraient être les moins coûteux. Toutefois, la connaissance de la sémantique de l'application se trouve en même temps à ce niveau car il y est possible de spécifier plus finement ce qui doit faire partie de l'état mobile de l'application et de minimiser ainsi l'état à transférer [Lawall et Muller, 2000].

Par ailleurs, rendre une application mobile peut éventuellement induire un surcoût sur les performances de l'application elle-même. Autrement dit, les traitements qui sont propres à une application, c'est-à-dire les traitements qui ne sont pas l'opération de mobilité, peuvent voir leurs performances se dégrader en raison de la prise en compte de la mobilité. Ce surcoût est évalué en comparant le temps nécessaire à l'exécution d'une application mobile, en dehors de l'opération de mobilité, et le temps nécessaire à l'exécution de cette même application lorsqu'elle n'est pas mobile. Un tel surcoût est, par exemple, induit par le mécanisme de la mobilité proposé par le système JavaGo [Sekiguchi et al., 1999] puisque ce dernier est basé sur une augmentation du code source de l'application mobile. Un surcoût sur les performances de l'application mobile est également induit par le mécanisme Java [Illmann et al., 2001] (dans CIA) car la mise en œuvre de ce mécanisme est basée sur le *debugger* Java ce qui induit un surcoût considérable sur l'application.

Quant au troisième coût, il représente l'éventuel surcoût induit par la mobilité sur les performances des applications non mobiles. Ce coût décrit la dégradation des performances des applications non mobiles même en l'absence d'application mobile s'exécutant sur le même

système. Ceci peut, par exemple, se produire à la suite de l'intégration d'un service de mobilité à un système existant. Cette intégration peut exiger la modification d'autres services du système. Cependant, une mise en œuvre modulaire permet généralement de séparer le service de mobilité des autres applications et réduire ainsi l'impact que peut avoir la mobilité sur les performances des applications non mobiles.

2.4.3 Réflexivité

La réflexivité est une des approches utilisées pour la migration de processus dans les systèmes hétérogènes. En effet, Attardi [Attardi et al., 1988] a utilisé la réflexivité pour mettre en œuvre la migration de processus dans des environnements hétérogènes. Selon [Smith, 1984] [Smith, 1982], la réflexivité désigne la capacité d'un système à s'auto-représenter et à se manipuler [Maes, 1987]. Un système réflexif assure un lien de cause à effet avec lui-même. Ce lien lui permet de s'observer et de se modifier de telle sorte que l'état de l'application est toujours cohérent avec sa représentation. Comme nous le montre la figure 2.18, l'architecture d'un système réflexif est composée d'un:

- *Niveau de base* qui est le lieu de traitement de l'information et en même temps responsable de l'aspect fonctionnel du système [Bennani, 2005]. Il offre des liens d'interactions pour acquérir les requêtes des utilisateurs et des liens d'interactions avec le méta-niveau.
- *Méta-niveau* qui implémente le calcul «réflexif» et qui peut observer et contrôler son comportement (réflexivité comportementale) et/ou sur sa structure (réflexivité structurelle) grâce aux mécanismes suivants [Ruiz-Garcia, 2002]:
    - *La réification*: Elle signifie exhiber (exposer) les caractéristiques des applications (niveau de base) de telle sorte qu'elles deviennent manipulables à un niveau d'abstraction plus élevé (le méta-niveau) en tant que données. Si la caractéristique réifiée est de nature événementielle (comme l'envoi d'un message), sa réification implique une *notification* explicite au méta-niveau grâce à laquelle ce dernier pourra suivre l'évolution de la dynamique du niveau de base. Lorsque la caractéristique réifiée est structurelle (comme la description du format d'un message), sa réification relève de la construction

d'une abstraction que le méta-niveau peut consulter au travers de mécanismes d'*introspection*.

- *L'intercession*: Le contrôle des caractéristiques «réifiées» et «introspectées» relève de l'intercession. On parle d'*intercession comportementale* lorsque la caractéristique ciblée par une action de contrôle est de nature comportementale (l'envoi effectif d'un message dont l'émission a été préalablement réifiée par exemple). Lorsque la manipulation à appliquer cible la structure ou l'état d'une entité du niveau de base (comme le changement de format ou de contenu d'un message), on parle alors d'intercession *structurelle*.

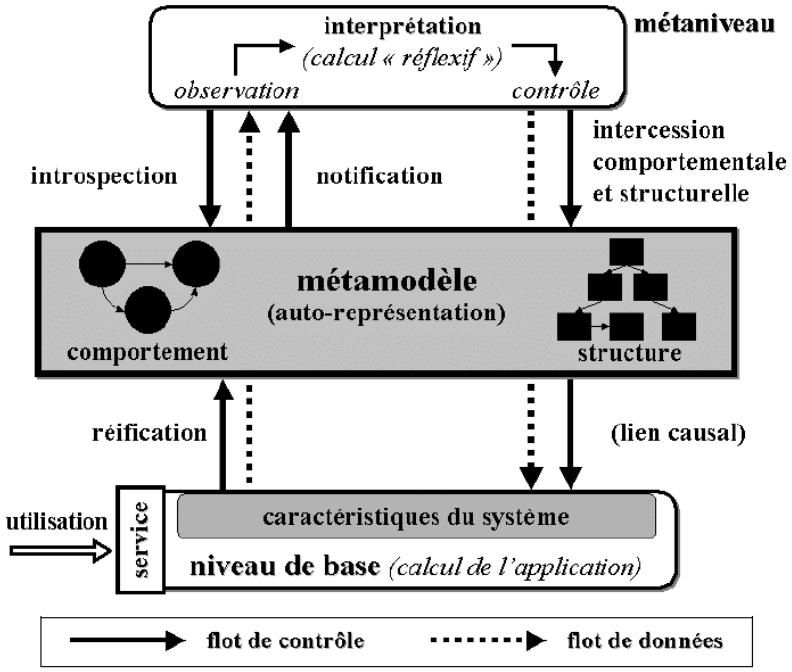


Figure 2.18 Vue générale d'une architecture réflexive [Ruiz-Garcia, 2002]

D'un point de vue extérieur, un système réflexif n'est pas plus puissant qu'un système qui ne l'est pas [Maes, 1987]. Tout problème résolu par une approche réflexive peut l'être d'une autre manière [Taïani, 2004]. Dans cette thèse, nous nous intéressons particulièrement l'aspect de la réflexivité qui permet la capture/restauration de l'état de programme (processus).

Réflexivité des langages interprétés

Un programme écrit dans un langage interprété, comme en LISP, en Perl ou encore en Java, est interprété par un programme spécifique appelé interprète [Taïani, 2004]. L'interprète évalue à l'exécution, le programme source et exécute son propre code machine. Un langage interprété *L* est réflexif s'il permet d'inclure dans un programme *P* écrit dans ce langage du code modifiant la manière dont *L* est interprété [Taïani, 2004]. La réflexivité offre des avantages mais elle introduit aussi des inconvénients aux langages interprétés. Certes, la réflexivité fournit un système très flexible, car tout est contrôlable à l'exécution du programme, mais cette dynamicité se paie en termes de performances [Killijian, 2000]. En effet, l'interprétation d'un programme est très coûteuse en temps processeur et d'espace mémoire. Il est pertinent de préciser que Java [Joy et al., 2000] possède certaines capacités réflexives mais celles-ci ne fournissent que des moyens d'observation relativement de haut niveau de processus d'interprétation du bytecode Java. Cependant, il ne permet pas de modifier l'interprète en redéfinissant, par exemple, le mécanisme d'appel.

A l'inverse des langages interprétés, les programmes écrits dans un langage compilé sont traduits en langage machine avant leur exécution. Le programme qui effectue cette traduction est appelé compilateur. Comme les langages interprétés réflexifs qui sont basés sur un interprète réflexif, les langages compilés réflexifs sont basés sur un compilateur réflexif. Celui-ci permet au méta-niveau de contrôler la façon dont le programme est compilé [Killijian, 2000]. Le méta-niveau peut alors être vu comme un ensemble de règles de traduction du programme source. On parle alors de compilateur ouvert, car le compilateur est programmable. Le méta-niveau est appelé aussi méta-programme. Il est responsable de la compilation du niveau de base, le programme source.

Au-delà du langage, un système peut être réflexif en termes d'environnement d'exécution. L'environnement d'exécution d'un système peut être considéré comme réflexif s'il fournit des moyens de contrôle à son propre comportement à un niveau supérieur (méta) en offrant donc un mécanisme de réification et d'intercession. Le fait qu'un environnement d'exécution soit observable et contrôlable, grâce à la réification et à l'intercession respectivement, l'ouvre au

monde extérieur et permet la capture/restauration de l'état d'exécution. Ceci est particulièrement important dans le contexte de la migration de processus.

2.4.4 Projet Attardi

Dans [Attardi et al., 1988], la migration de processus par interprétation ou traduction est mise en œuvre à l'aide de la réflexivité. Dans ce contexte de migration de processus, la réflexivité utilise le mécanisme de *réification* et de *réflexion*.

Migration de processus par interprétation

Dans l'approche de l'interprétation, le même interprète s'exécute sur la machine source et sur la destination. Ceci revient à dire qu'il existe une machine universelle qui émule les deux ordinateurs. Le processus à migrer est mis sous forme d'une représentation indépendante de la machine. L'exécution du programme/processus consiste à interpréter la représentation indépendante de la machine. La migration consiste à communiquer la représentation à travers le réseau. La construction du mécanisme de migration implique la mise en œuvre des interprètes adaptés dans chaque machine et fournit un mécanisme de communication d'inter-machine approprié.

Dans l'approche de migration de processus via interprétation, une machine intermédiaire connue sous l'abréviation de *HLACM (High Level Abstract Common Machine)* [Attardi, 1987] est définie. *HLACM* est dérivée de la machine de Chaos de L. Cardelli [Cardelli, 1984] [Cardelli, 1983]. La machine universelle de haut niveau d'abstraction est essentiellement un interprète du *bytecode*. C'est une machine orientée-pile conçue pour supporter des langues fonctionnelles de haut niveau. Celle-ci fournit un mécanisme de mémoire indépendant de la machine.

HLACM fournit deux primitives: *réification* et *réflexion*. La *réification* s'opère du niveau de base vers le méta-niveau lorsque le premier subit une modification devant être reflétée dans le second. Elle fournit une représentation du processus (programme) en cours et de son état d'exécution. La *réflexion*, quant à elle, va du méta-niveau vers le niveau de base lorsqu'une

modification du premier doit s'opérer sur le second. La primitive *réflexion* crée un nouveau processus avec la représentation reçu en argument.

Chaque hôte du réseau doit disposer d'un interprète *HLACM* qui exécute un ensemble de *processus* géré par un ordonnanceur. Les *processus* sont fondamentaux dans la phase de migration. En fait, ils permettent de construire des schémas organisationnels de migration. Un des *processus* du système, connu sous le nom *serveur de migration*, assure la communication avec les autres hôtes sur le réseau en écoutant continuellement les demandes de connexion. Une demande est publiée en transmettant la requête au *serveur de migration*. Lorsque le serveur de migration reçoit une telle demande, il l'analyse et exécute l'action correspondante.

Lorsqu'un processus doit migrer vers un nouveau site pour l'exécution, le *processus serveur* reçoit la requête, il crée un nouveau *processus* et l'exécute dans le nouveau contexte. Une représentation indépendante de la machine est utilisée pour la migration des processus vers les autres machines.

HLACM résout le problème de migration de processus entre les machines de différentes architectures à l'aide d'une couche intermédiaire interprétée. Cette approche est intéressante mais peut présenter des inconvénients: l'efficacité (temps de migration), devient, d'abord, critique pour les grosses applications mais aussi la gamme de programmes, auxquels ces mécanismes de migration s'appliquent, demeurent limités. En effet, un programme peut migrer seulement s'il est compilé dans HLACM.

Migration de processus par traduction

Dans l'approche de la traduction, le programme/processus à migrer est mis dans un format qui dépend de la machine. Avant la migration, le processus doit être traduit en une description indépendante de la machine. Cette description peut ensuite être communiquée à la destination afin de poursuivre son exécution. La construction du mécanisme de migration implique la mise en œuvre des traducteurs appropriés sur chaque machine.

La machine *HARP (*abréviation *pour Harlequin Abstract RISC Processor)* [Hunter et Knightbridge, 1988] [Knightbridge et Hunter, 1987] est conçue afin de fournir un support particulier au mécanisme de migration de processus basé sur la traduction. HARP joue le rôle d'un formalisme indépendant de représentation de la machine dans laquelle tout état d'un processus peut être capturé. Le développement du modèle prend en compte la condition de traduction entre les descriptions de HARP et les représentations réifiées d'un hôte dépendant. Il est pertinent de mentionner qu'il existe une grande ressemblance entre la machine HARP et les processeurs les plus couramment utilisés dans le but de rendre cette traduction aussi simple que possible tout en conservant un niveau d'abstraction.

Chaque contexte de HARP représente un processus d'exécution. Le contexte et l'unité de manipulation de contexte permettent à la machine HARP de modeler le routage et fournissent un mécanisme de signal.

Lors de migration d'un processus d'une machine à une autre, HARP convertit toutes les adresses en références, puis en étiquettes lors de l'étape de la réification. Lors de la réflexion, les adresses doivent être récupérées. Ce mécanisme exige que le code de réification/réflexion puisse toujours distinguer entre les données et les pointeurs. Pour répondre à cette exigence, le système de marquage suivant est inclus dans la définition:

- Chaque bloc de mémoire est marqué selon qu'il contient un code ou des données;
- Chaque registre peut être individuellement étiqueté selon qu'il contient une donnée ou un pointeur. Les opérations empiler/dépiler transfèrent les informations de marquage de/vers l'élément supérieur de la pile de/vers le registre concerné.

La définition HARP évite toute restriction de la représentation locale du code. L'absence d'une syntaxe stricte sur le code réflexif permet aux traducteurs de remplacer des instructions de HARP du code natif équivalent indépendamment de sa taille. Rappelons qu'un code natif (ou langage machine) est composé d'instructions directement reconnues par le processeur sous-jacent.

Un registre global contenant un pointeur vers le début du prochain bloc de mémoire est attribué au processus en cours d'exécution. Ce mécanisme d'allocation de mémoire permet au code de HARP de créer de nouveaux blocs contenant du code ou des données. Ceci est essentiel pour la réflexion.

Une représentation du processus HARP prend la forme d'un bloc de code qui, lorsqu'il est exécuté, va recréer le processus initial. L'avantage de cette approche est que la représentation ait un contrôle explicite du processus de chargement. Ceci assure non seulement le rôle d'un chargeur dans les systèmes plus classiques mais également contient le programme/données à charger.

2.4.5 Système Tui

Le système Tui [Smith et Hutchinson, 1997] permet de migrer de programmes écrits en langage C au cours de leur exécution entre différentes architectures. La conversion de données est plus complexe puisqu'il faut tenir compte de difficultés comme la mauvaise utilisation de pointeurs, la conversion (ou *casting* en anglais) de type et le manque d'informations de type explicites. Moins le langage est sécurisé, plus il devient difficile de localiser et assigner à un type de données. Ces problèmes n'apparaissent pas dans les langues de type-sécurisé.

Bien qu'on dise que les langages de programmation non typées ont tendance à générer des programmes «non-transférables», il est nécessaire d'aborder chaque problème sur une base individuelle. Par exemple, un programme écrit en C peut être «non-transférable» en raison de la façon dont une petite partie du programme a été écrit. La réécriture de cette section de code d'une manière différente permettra la migration. L'exemple de code en C de la figure 2.19 nous le démontre [Smith et Hutchinson, 1997].

```
{
        union data
        {
                int a;
                float b;
        }myData;
                myData.a = 1;
                myData.b = 23.45;
}
```

Figure 2.19 Exemple de code en C à sécuriser

Lors de la migration de ce programme, le système doit identifier l'attribution la plus récente à l'union. Le système traduit soit le nombre entier 1 ou le nombre à virgule flottante 23.45. Étant donné qu'ils partagent le même emplacement de mémoire, le système de migration ne peut pas interpréter les données. Dans ce cas, plusieurs approches sont possibles:

- Modifier le compilateur pour maintenir l'élément de l'*union* qui a été le plus récemment accédé;
- Réécrire le code en utilisant *struct* au lieu d'*union* ainsi les éléments ont des emplacements de mémoire distincts;
- Le programmeur doit s'abstenir d'utiliser le type *union*.

Dans Tui, le système de migration de processus est capable de migrer de programmes ANSI-C de type sécurisé entre quatre architectures: Solaris s'exécutant sur un processeur SPARC (Sun 4), SunOS sur un m68020 (Sun 3), Linux sur un i486 et un AIX sur un PowerPC. La figure 2.20 nous montre la migration d'un processus dans le système Tui. La migration d'un processus dans le système Tui suit les étapes suivantes:

1. Le programme (écrit en *C ANSI*) est compilé, une fois pour chaque architecture. Une version modifiée du kit de compilateur d'Amsterdam (en anglais Amsterdam Compiler Kit, ACK) [Tanenbaum et al., 1983] peut produire des fichiers binaires pour chacun des quatre types des machines supportées par le système Tui.
2. Le programme est exécuté sur la machine source de façon standard (par exemple, à partir de la ligne de commande).
3. Lorsque le processus a été sélectionné pour la migration, un programme *migrout* est appelé au poste de contrôle de ce processus. À l'aide de l'identité du processus et le nom du fichier exécutable (contenant des informations de types), le programme

migrout lit l'espace mémoire utilisé par le processus et analyse les variables globales, la pile et le tas pour localiser toutes les valeurs de données. Enfin, toutes ces valeurs sont converties en un format intermédiaire. Elles sont envoyées ensuite à la machine destinataire.

4. Sur la machine destinataire, un programme *migrin* prend la représentation intermédiaire et crée un nouveau processus. On suppose que le programme soit compilé pour l'architecture cible de sorte que le segment de texte complet et les informations de types pour le segment de données soient disponibles. Après la reconstruction des variables globales, le tas et la pile, le processus est repris du même point d'exécution qu'il avait été arrêté sur la machine de départ.

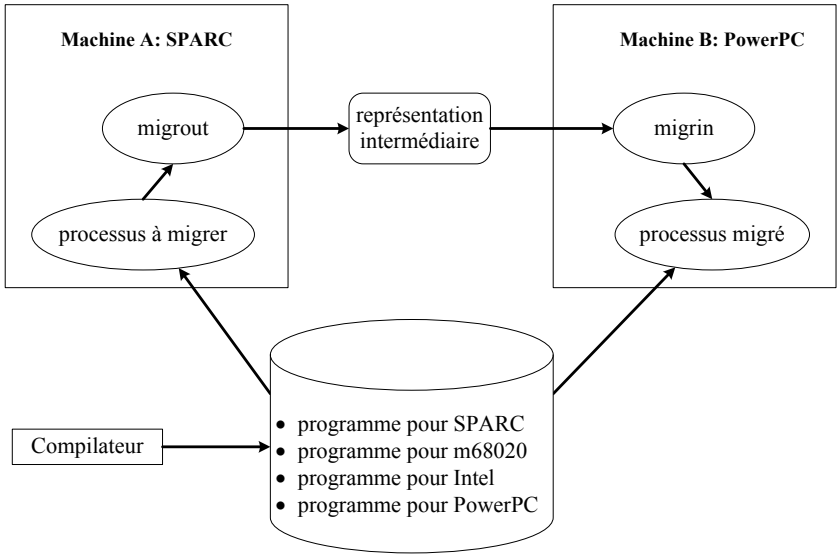


Figure 2.20 Migration d'un processus dans le système Tui [Smith et Hutchinson, 1997]

La structure d'ACK s'est avérée bien adaptée à la génération de code «transférable». Il est souhaitable qu'un programme exécutable ait exactement la même structure sur toutes les machines cibles. Autrement dit, chaque programme est compilé pour contenir le même ensemble de symboles (procédure et noms de variables) et chaque procédure contient le même ensemble de variables locales et intermédiaires. L'emplacement de stockage et la taille de ces entités peuvent être très différents entre les machines. Par exemple, les variables locales peuvent être stockées sur la pile ou dans les registres.

ACK génère les informations de débogage en format stabs [Menapace, 1993]. Ces informations décrivent le type et l'emplacement de toutes les valeurs de données en utilisant un codage d'ASCII. En plus, la correspondance entre les numéros de ligne de code source et les adresses de machine cibles est enregistrée. Cette information est généralement utilisée par les outils de débogage pour permettre au programmeur d'étudier les valeurs des données d'un programme en cours d'exécution. Le système Tui utilise ces valeurs d'une manière similaire, mais plus automatique.

Les informations de types de base utilisées par les débogueurs ne sont pas suffisantes pour migrer correctement un programme. Certains ajouts sont nécessaires pour traduire toutes les valeurs de données. Nous allons examiner les trois principaux ajouts: les points de préemption, les points d'appel et la taille de pile.

Points de préemption

Lorsqu'un processus est transféré, on doit composer avec le fait que le point d'arrêt d'exécution correspondant (compteur ordinal) aura un emplacement différent dans des systèmes d'architectures variées. Pour résoudre ce problème, il faut sélectionner un ensemble de points de migration de processus. En exécutant le programme *migrout* au point de capture d'état d'un processus, on doit veiller que l'exécution s'arrête à l'un de ces points de préemption. Lors du redémarrage du processus, la valeur correcte du compteur ordinal peut être déterminée. En effet, le programme doit avoir un ensemble identique de points de préemption sur chaque architecture cible.

Le placement de points de préemption dans un programme est une question intéressante. Les points doivent être placés plusieurs endroits pour que le processus s'arrête en minimum de temps (excluant la possibilité des appels de système qui pourraient bloquer). Par exemple, si un point de préemption pour un processeur *SPARC* est placé dans une séquence d'instructions qui effectuent une opération de multiplication, il n'y a aucun moyen de placer le point correspondant dans le programme sur un processeur *VAX* puisque ce dernier ne nécessite qu'une seule instruction pour effectuer la multiplication. En tenant compte de ces limitations, les points de préemption sont placés au début de boucle et à la fin de chaque instruction

composée. En outre, on permet à l'optimiseur cible de chaque machine de manipuler n'importe quel code dans un bloc de base, mais il ne doit pas déplacer le code à travers des points de préemption.

Points d'appel

Bien que le placement minutieux des points de préemption permette de minimiser le nombre de valeurs intermédiaires (résultats partiels d'un calcul), elles peuvent toujours se trouver entre les appels de procédure. L'exemple suivant illustre cette situation: *x = foo(y) + bar(y)*. Dans ce fragment de code, le résultat de *foo(y)* doit être sauvé quelque part durant l'exécution de *bar(y)*. Cependant, si le processus est interrompu pendant l'appel de *bar(y)*, il est nécessaire de récupérer la valeur de *foo(y)* de son emplacement intermédiaire (dans la pile ou dans un registre). Lors de la reconstruction du processus dans la machine cible, la valeur intermédiaire sera restaurée.

Ce résultat est obtenu en générant un *stabs* de point d'appel à chaque appel de procédure. Il spécifie l'adresse de l'instruction d'appel, le nombre d'intermédiaires (expressions partiellement évalués), le nombre de paramètres passés, le type et les informations de l'emplacement de chacune de ces valeurs.

Taille de pile

Lors du processus du programme *migrin*, le système Tui doit reconstruire chaque trame de pile qui existait avant la migration. Au moment de la compilation, une chaîne *stabs* spéciale est créée au début de chaque procédure. Celle-ci spécifie la taille de la trame de pile (le nombre d'octets utilisés pour obtenir les informations comme les variables locales), ainsi que les registres qui ont été sauvés sur la pile lors de l'entrée à cette procédure.

Migrout: Points de reprise du processus

La description du processus de *migrout* est divisée en quatre phases principales. Premièrement, le type et les informations d'emplacement (produit par le compilateur) sont mis dans des structures de données internes du système Tui. Ensuite, le processus à migrer est

interrompu et son image mémoire est copiée dans l'espace d'adressage du système Tui. Troisièmement, les informations de type sont utilisées comme un guide pour parcourir cette mémoire et localiser toutes les valeurs de données. Finalement, ces valeurs sont traduites dans un format intermédiaire pour la transmission vers le programme *migrin* du système Tui.

Informations de types

Les informations de débogage *stabs* associées au programme sont spécifiées d'une manière qui suit la structure du programme. La table des symboles du fichier exécutable contient une section pour chaque fichier objet (fichier.o) qui constitue l'exécutable. Dans chaque section, les variables globales et les procédures sont énumérées avec leur type et les informations de localisation. Pour les procédures, la même information de types est donnée pour les paramètres locaux et intermédiaires.

Les informations de débogage de format *stabs* sont converties (à la compilation) dans des structures de types appropriées. Ces structures, connues sous le nom des arbres de types, sont similaires à celles utilisées à la plupart des compilateurs. Elles peuvent représenter tous les types de base ainsi que les pointeurs, les tableaux et les structures. Pour éviter les conflits de noms, chaque symbole est préfixé avec le nom de son fichier et les valeurs locales avec le nom de la procédure.

En outre, deux tables supplémentaires sont nécessaires. La première table enregistre les points de préemption où chaque entrée contient une adresse unique. La seconde table effectue une opération similaire, mais pour les points d'appel. Dans les deux cas, l'indice de la table est utilisé comme une représentation indépendante de la machine de l'adresse du point.

L'arrêt d'un programme est plus complexe que la migration sur des machines homogènes. L'appel du système *Unixptrace* est utilisé pour placer le processus dans l'état de trace. Le programme *migrout* peut maintenant faire une copie de la mémoire et les registres. Cependant, il faut s'assurer que le processus soit dans un état cohérent (à un point de préemption). Le code exact de l'application dépend de la machine.

Capture de l'image du processus

En parcourant la mémoire du processus pour localiser toutes les valeurs de données, il faut s'assurer que chaque valeur soit lue exactement une fois. Ceci est fait en maintenant une table de valeurs qui enregistre l'adresse de départ, la taille et le type de données. La table est mise en œuvre à l'aide d'une structure de données extensible où on ajoute chaque nouvelle valeur à la fin. Ainsi, la mémoire est parcourue d'une façon linéaire pour que les valeurs soient ajoutées à la table dans le bon ordre.

Premièrement, les points d'entrée de procédure et les variables globales sont parcourus et leurs contenus sont entrés dans la table de valeurs. Les variables globales sont très simples à traiter puisque leurs emplacements sont fixes et leurs types sont bien définis.

Pour parcourir les données de tas dans l'ordre, il est nécessaire de modifier les procédures d'allocation (*malloc*) et de libération *(free)* de mémoire pour qu'ils puissent enregistrer tous les blocs (vides et utilisés) dans l'ordre. Cet ajout coûte un pointeur supplémentaire par bloc de mémoire. De plus, le compilateur doit produire un (petit bout de) code supplémentaire pour enregistrer le type de données de chaque bloc qui est alloué.

Les variables locales (contenu dans la pile) sont parcourues d'une façon semblable. La pile est parcourue à partir du sommet. À chaque point, Tui interroge les informations de type du programme afin d'obtenir la liste de procédure de la pile ou le contenu des registres. Lorsque les valeurs stockées dans la pile sont spécifiées comme des décalages *(offsets)* de pointeur de procédure, les adresses absolues doivent être calculées. Une attention particulière est également portée à l'ensemble de registres courants, d'autant plus qu'ils sont souvent sauvegardés sur la pile pendant les appels de procédure.

À chaque fois qu'un appel de procédure est effectué, Tui localise le point d'appel associé pour déterminer les données intermédiaires et les arguments qui ont été stockés dans la pile pendant la durée de l'appel. Les arguments d'une procédure seront parcourus dans la perspective de l'appelant pour gérer correctement les procédures qui permettent à un nombre variable d'arguments.

Finalement, les arguments de ligne de commande et les variables d'environnement sont parcourus. Ceci doit être fait séparément de pile puisque ces informations ne sont pas toujours décrites par un nom de variable explicite comme une variable de pile normale.

Mise en forme intermédiaire

L'étape finale de *migrout* est de parcourir le tableau de valeurs et encoder toutes les valeurs de données provenant de la mémoire du processus dans le fichier intermédiaire. Ceci exige la conversion de format de données comme une conversion de formats d'entiers de petit-boutiste (*little endian)* à gros-boutiste (*big endian*) par exemple. La difficulté se situe au niveau de la représentation de la relation entre les éléments de données différents. Autrement dit, certaines données contiendront des pointeurs vers d'autres. Lors de mise en forme d'une valeur de pointeur, le système Tui effectue une recherche binaire sur la table de valeurs pour localiser les informations relatives à l'objet pointé.

Chaque entrée de la table de valeurs est assignée à un nombre unique. Lorsqu'une référence est faite à un élément de données, le pointeur est codé en spécifiant ce nombre indépendant plutôt que l'adresse spécifique de la machine. En outre, dans le cas où un pointeur se réfère à un emplacement partiel par un élément de données composite, un décalage (offset) indique le nombre de sous-éléments indivisibles qui doivent être ignorés afin de localiser la valeur correcte. Le code écrit en C de la figure 2.21 nous présente un exemple de de cette situation [Smith et Hutchinson, 1997]:

```
{
        struct
        {
                int a;
                int b;
        }c[10];
        int *p = &c[2].b;
}
```

Figure 2.21 Code écrit en C pour décrire le concept de décalage *(offset)*

Dans le cas de ce code, le décalage du pointeur *p* serait 5 puisque la structure contient deux sous-éléments et *p* se réfère au deuxième élément de la troisième instance de cette structure dans le tableau *c*.

Migrin: Reconstruction de l'image du processus

Pour redémarrer un processus dans la machine de destination, l'algorithme *migrin* doit obtenir le type du programme et les informations d'emplacement de la même manière que le programme *migrout*. Ensuite, il parcourt le fichier intermédiaire et met toutes les valeurs de données dans leurs emplacements appropriés. Cette phase lit le fichier intermédiaire séquentiellement et peut donc surtout être faite en parallèle avec la phase de *migrout*.

Les variables globales sont mises directement dans leurs emplacements de mémoire absolus. La pile virtuelle et les pointeurs de tas sont maintenus avec toutes les nouvelles valeurs étant ajoutées à la fin du segment approprié. Il est essentiel que les éléments de données sur la pile soient restaurés dans le bon ordre. Également, en raison du mode linéaire suivant lequel la table de valeur est construite pendant la phase de *migrout*, le tas doit maintenir l'ordre correct.

Les pointeurs causent aussi des problèmes en plaçant des valeurs de données dans la mémoire. Il n'est pas possible de déterminer la valeur finale d'un pointeur jusqu'à ce que l'objet, auquel il se réfère, ait été assigné à un emplacement de mémoire. Par conséquent, une table est utilisée pour enregistrer tous les pointeurs et une fois toutes les valeurs de données traitées, les pointeurs sont convertis de leurs *(Object ID, offset)* paires dans des adresses de machine.

Le processus est repris en chargeant le fichier binaire du programme dans la mémoire. Les données nouvellement construites et les segments de pile sont écrits dans l'espace d'adresse (en utilisant *ptrace*). Ensuite, le point de préemption, qui représente l'adresse de continuation du processus, est converti correctement en adresse dépendante de la machine. Enfin, on donne les valeurs correctes de registre au processus et il continue l'exécution.

Représentation intermédiaire

Le fichier intermédiaire est une représentation indépendante de la machine de la table de valeurs. Il répertorie toutes les valeurs de données dans un format de stockage bien défini et, si nécessaire, déclare le type de valeurs et la relation entre elles. Toutes les valeurs de données (*entier* et *flottant*) sont codées en utilisant le format de stockage natif pour les machines Sun 4, c'est-à-dire les entiers en complément à deux en *big endian* et en virgule flottante IEEE. Comme Sun 3 et les PowerPC utilisent également ce format, le processeur Intel est la seule machine qui a besoin de faire une conversion de format.

Les éléments de données sont répertoriées dans l'ordre: les procédures, les variables globales, les valeurs de tas et de la pile. Voici l'ordre dans lequel les éléments apparaissent dans l'espace d'adresse d'architectures supportées par le système Tui:

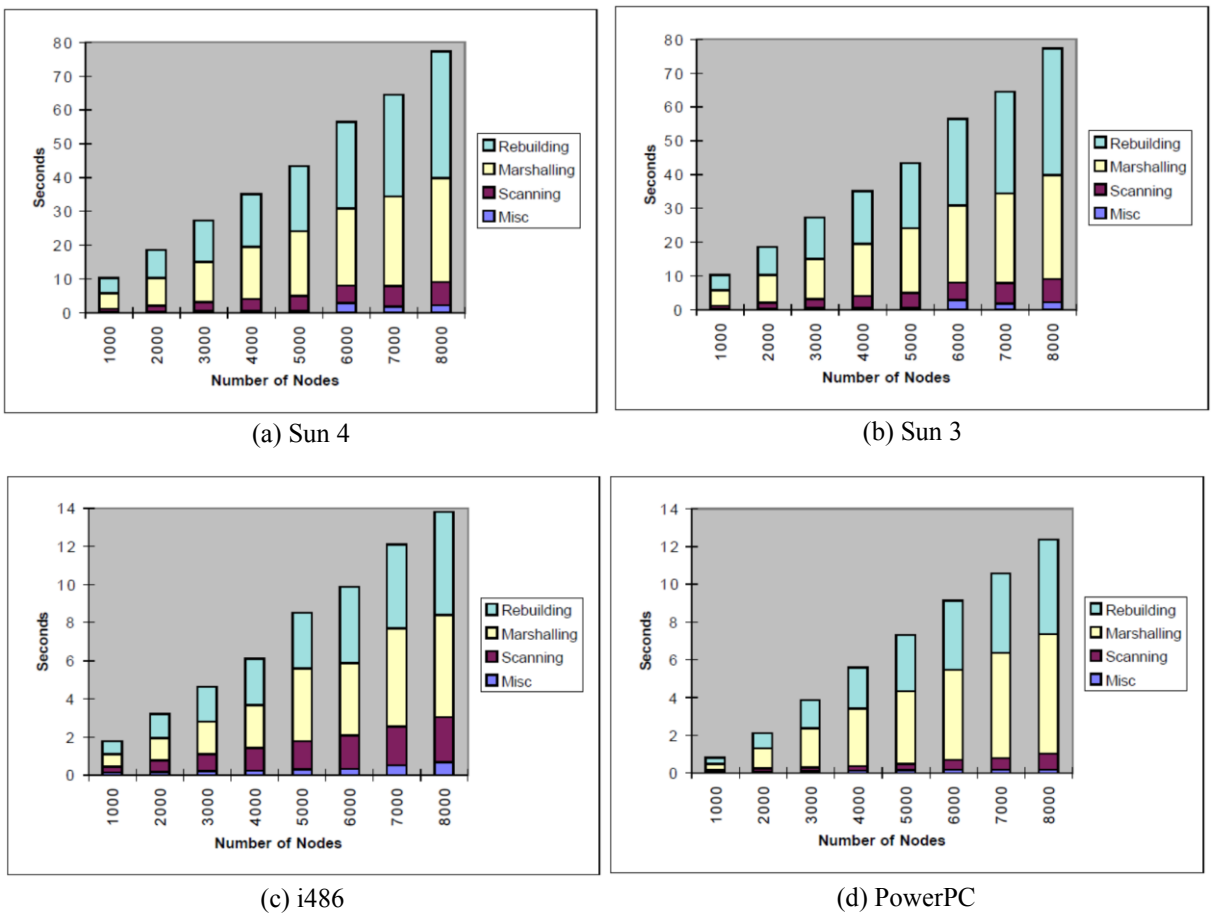
- *Les Procédures*: le nom de chaque procédure est inscrit, puisqu'il est possible pour un pointeur de se référer à une procédure. Aucune autre information n'est donnée sur le segment de texte.
- *Les variables globales*: le nom de la variable et sa valeur sont spécifiés. Il est nécessaire d'inclure le nom, car les variables peuvent apparaître dans un ordre différent sur les diverses architectures. En outre, certains symboles peuvent exister sur une machine, mais pas sur l'autre. Ces valeurs seront généralement dépendantes de la machine. Elles ne sont normalement pas utiles et il n'est pas nécessaire de migrer.
- *Les valeurs de tas*: Celles-ci ne portent pas de nom et la machine de destination ne peut pas déterminer le type de données à l'avance. Par conséquent, les valeurs sont répertoriées avec leur numéro de type *stabs*. Il est nécessaire que toutes les architectures utilisent un système de numérotation de même type.
- *Les valeurs de la pile*: Elles sont énumérées dans leurs trames respectives. Chaque trame est identifiée par le nom de la procédure et le numéro du point d'appel qui a créé la trame. Les paramètres, les variables locales intermédiaires et les arguments sont inscrits dans un ordre qui soit compatible entre les machines. Les noms des variables ne sont pas nécessaires.

Évaluation de performances

Pour évaluer les performances des algorithmes de Tui, trois programmes test ont été créés. Chacun est conçu pour tester la complexité des différentes composantes de Tui. Les trois programmes sont:

1. *Fibonacci* dans lequel une mise en œuvre récursive de l'algorithme *Fibonacci* crée un nombre de trames de pile où chacune a un nombre des variables locales et intermédiaires. Un seul point de préemption est placé de telle sorte que la migration se produit lorsque N trames de pile sont actives (N est le paramètre d'entrée). Ce programme teste l'efficacité du programme *migrout* en parcourant la pile.
2. *Arbre* construit un arbre binaire de N nœuds. Des valeurs numériques sont choisies aléatoirement et y sont ensuite insérées. Une fois que la construction de l'arbre est complétée, la migration va se produire. Ce programme teste la capacité de Tui de parcourir l'espace de tas dans un ordre aléatoire.
3. *Tableaux*: 50 tableaux de caractères (l'utilisateur spécifie la taille) sont alloués dynamiquement sur le tas, puis remplis de caractères. Ce test démontre l'efficacité du codage et de la reconstruction des grandes zones de mémoire.

Pour démontrer la bonne fonctionnalité de Tui sur les quatre architectures supportées, chacun des programmes a été migré. La figure 2.22 illustre les temps utilisés par les composants principaux (*migrin* et *migrout*) pour la migration du programme *arbre*. Dans ces tests, le nombre de nœuds de l'arbre varie de 1000 à 8000. Bien que seulement le programme *arbre* soit analysé, les résultats de *fibonacci* et *tableaux* étaient similaires [Smith et Hutchinson, 1997]. Les quatre machines utilisées pour ce test sont: Sun 4/75 (SPARCStation 2), Sun 3/60, i486 à 50Mhz et PowerPC 601 à 66 Mhz.



(a) Sun 4 (b) Sun 3

(c) i486 (d) PowerPC

Figure 2.22 Temps utilisés par les composants pour la migration du programme arbre [Smith et Hutchinson, 1997]

Les machines sont équipées d'une mémoire suffisante pour éviter la pagination. Dans cette analyse, le temps d'exécution total est divisé en deux composantes principales: *migrout* et *migrin*. Nous devons prêter attention aux coûts relatifs entre les composants et la croissance de chaque composante avec l'augmentation du nombre de nœuds. Voici les différents composants qui constituent le coût de migration du programme:

- *Divers (Miscellaneous)*: c'est le temps nécessaire pour lire l'image mémoire du programme de migration dans l'espace d'adressage Tui ainsi que le temps pour lire les informations de type du programme sur le disque. La taille de l'image de mémoire varie en fonction de la taille du programme mais la quantité d'informations de type reste constante.

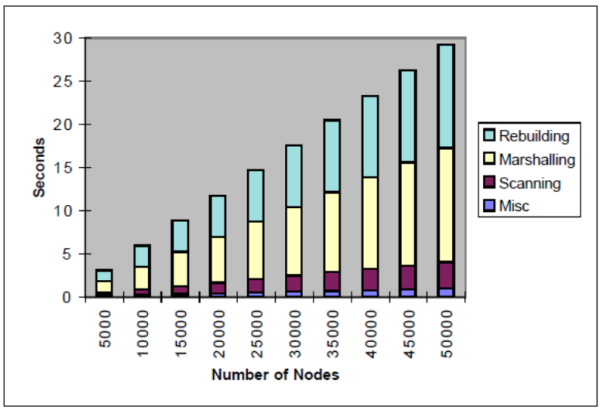
- *Balayage (Scanning)*: Comme son nom l'indique, il s'agit du balayage des segments de mémoire et de la construction de la table de valeurs. Son coût dépend du nombre de valeurs de données individuelles qui sont placées et non de la taille de ces valeurs.
- *Codage (Encoding)*: Les valeurs des données doivent être rassemblées dans le fichier intermédiaire. Son coût dépend de la taille totale de toutes les valeurs de données et de la performance du système d'exploitation lors de l'écriture des fichiers.
- *Reconstruction (Rebuilding)*: C'est la seule composante de l'algorithme *migrin* qui a été analysée. Compte tenu du fichier intermédiaire, les nouvelles données et les segments de pile sont construits. Les autres composants de *migrin* tels que la lecture et les informations de types de lecture/écriture de mémoire de base est la même que ceux *migrout*. Attention, la reconstruction finale de *migrin* peut se produire presque entièrement en parallèle avec le balayage et le codage de la phase *migrout*. Par conséquent, le temps de migration totale sera inférieur à la durée totale nécessaire sur tous les composants.

On peut constater que le *balayage*, le *codage* et la *reconstruction* sont les principaux composants qui ont les plus hauts coûts de migration. Les coûts *divers* de la lecture des informations de type et l'image de mémoire sont minimes et pourraient être négligés. On remarque que le coût des trois principaux composants augmente en fonction de la taille de l'entrée.

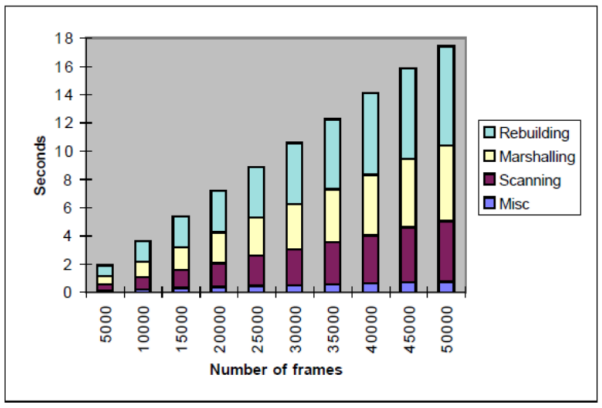
Pour examiner la performance de Tui en migrant des programmes, chacun des trois tests a été configuré pour qu'il crée une grande image de mémoire. La figure 2.23 montre la contribution des principaux coûts (le balayage, le codage et la reconstruction) en fonction du nombre de nœuds. Pour éviter les problèmes de pagination, tous les tests ont été effectués sur une même machine de grande taille *(PowerPC)*. Nous constatons que:

- Le programme *arbre* a presque une performance linéaire pour les trois composants. Les résultats du *balayage* et de la *reconstruction* font du sens mais pour le *codage* on s'attendait à une complexité légèrement plus haute en raison de la recherche binaire qui est faite sur la table de valeur pour chaque indicateur.

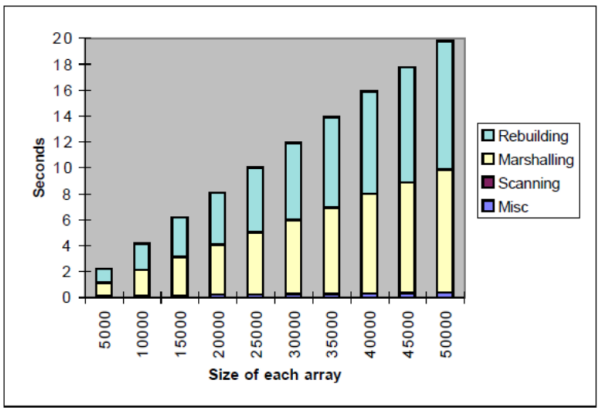
- Le programme *fibonacci* a à peu près la même complexité que le programme *arbre* mais la durée globale est plus faible.
- Le programme *tableaux*: Comme il y a seulement 50 tableaux, le composant de balayage nécessite une quantité non négligeable de temps pour les localiser. Toutefois, étant donné que chaque tableau peut être important (jusqu'à 50000 caractères dans ce cas-ci), le codage et les composants de reconstruction sont significatifs bien qu'ils aient toujours une complexité linéaire.



(a) Croissance du programme *arbre*



(b) Croissance du programme *fibonacci*



(c) Croissance du programme *tableaux*

Figure 2.23 Contribution des principaux coûts en fonction du nombre de nœuds [Smith et Hutchinson, 1997]

2.4.6 Migration de processus dans la machine virtuelle Java

La machine virtuelle Java offre des services de sérialisation et dé-sérialisation qui permettent respectivement de capturer et de restaurer l'état d'un objet Java. Ces services peuvent être utilisés pour déplacer des objets entre différentes machines. De plus, la machine virtuelle Java offre un service de chargement dynamique des classes qui permet de déplacer du code Java vers d'autres nœuds. La figure 2.24 nous le présente l'architecture de la machine virtuelle Java [Quang La, 2008].

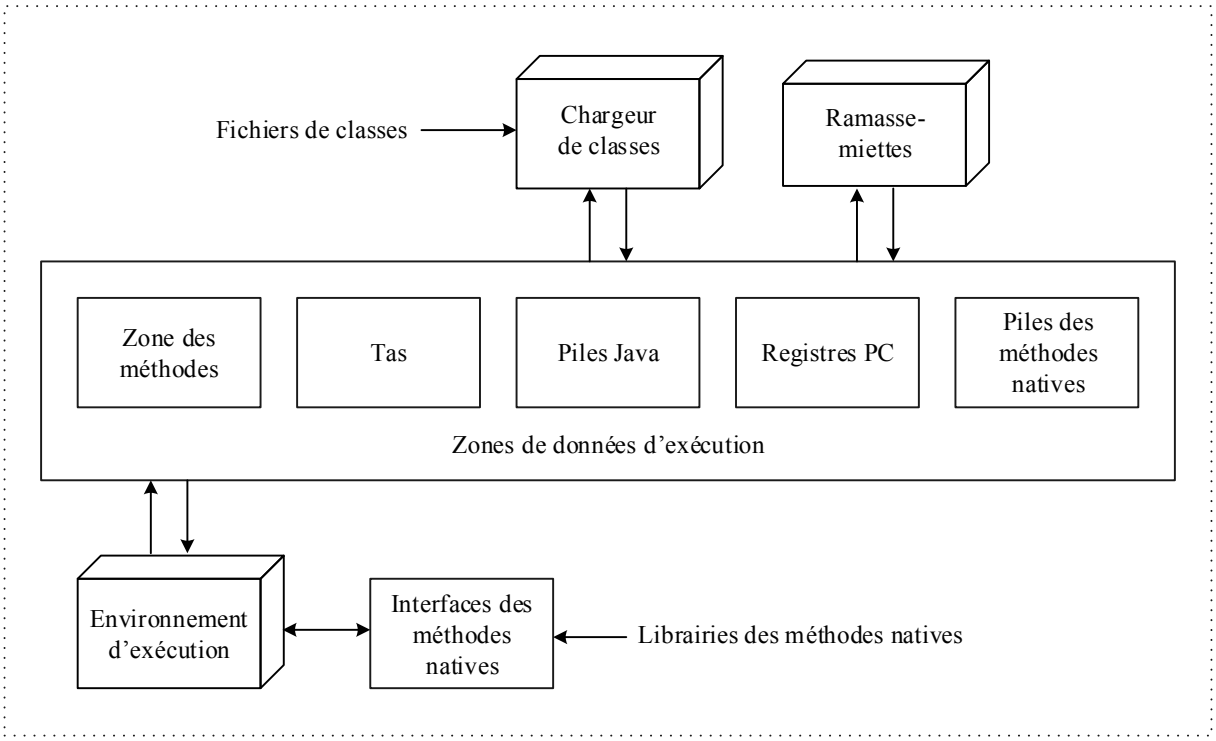


Figure 2.24 Architecture de la machine virtuelle Java

En revanche, Java ne fournit pas de service permettant de capturer et de restaurer l'état d'un flot de contrôle (c'est-à-dire un processus) car sa pile n'est pas accessible. Un tel service devrait permettre d'une part d'extraire l'état courant de l'exécution du flot et, d'autre part, de restaurer cet état pour que l'exécution reprenne au point où elle en était au moment de la capture. Autrement dit, ce service doit permettre la mobilité forte de l'application. Pour fournir un tel service, les auteurs proposent [Bouchenak et al., 2004] [Bouchenak, 2001]:

1. La conception d'une machine virtuelle Java étendue qui prend en charge la migration de processus Java avec les propriétés suivantes:
    a. La syntaxe du langage Java n'est pas modifiée;
    b. Le compilateur Java n'est pas modifié;
    c. Les APIs Java existantes ne sont pas affectées;
    d. La nouvelle API Java est proposée pour un mécanisme de sérialisation de processus générique;
    e. L'API Java de haut niveau pour la mobilité se trouve au-dessus de la couche de sérialisation de processus.
2. La mise en œuvre d'un mécanisme de sérialisation de processus Java sans aucune baisse de performances. Cette mise en œuvre est principalement basée sur deux techniques:
    a. Une inférence de types;
    b. Une dés-optimisation dynamique du code Java compilé.

Les travaux dans [Bouchenak et al., 2004] [Bouchenak, 2001] se concentrent sur la conception et la mise en œuvre d'un mécanisme de sérialisation de processus Java au-dessus duquel la mobilité et la persistance d'état d'exécution sont construites. Il est pertinent de préciser comme pour la sérialisation d'objets Java, la sérialisation de processus permet à un état d'un processus d'être transmis à un ordinateur distant afin de le restituer. Cependant, contrairement à la sérialisation d'objet, la sérialisation de processus ne traite pas la distribution, le partage d'objet entre les processus, la synchronisation et la gestion des objets d'entrées/sorties (les sockets ou les fichiers).

La sérialisation de processus est un mécanisme de base utilisé pour la mise en œuvre d'un environnement intergiciel (*middleware*). La figure 2.25 illustre comment la sérialisation de processus a lieu dans un tel intergiciel [Bouchenak et al., 2004]. À titre d'exemple, prenons les trois systèmes suivants:

1. Le système d'agents mobiles. Dans un tel système, les agents sont généralement encapsulés dans des conteneurs d'objets Java qui migrent en utilisant la sérialisation d'objets [Bouchenak et al., 2004]. La sérialisation de processus peut donc être utilisée

à la place de la sérialisation d'objets dans le but de transformer la mobilité faible des agents en mobilité forte. Dans le système d'agents mobiles *Aglets* [Lange et Oshima, 1998], les interactions entre les agents sont basées sur des échanges de message et l'objet Java partageant la gestion est ainsi évité. Une description détaillée sur l'isolement des applications Java est présentée dans [Czajkowski, 2000].

2. Les systèmes *Ajents* [Izatt et al., 2000] et *Javanaise* [Hagimont et Boyer, 2001] utilisent le mécanisme de partage d'objet. *Javanaise* est un système d'objets distribués répliqué où la synchronisation de répliques est basée sur un protocole de cohérence d'entrée [Bershad et al., 1993]. Un tel système pourrait être combiné avec la sérialisation de processus pour construire un service de migration de processus distribué complet qui bénéficie de la réplication et la synchronisation.
3. Le système distribué responsable de la gestion des objets d'entrées/sorties. Accent/Mach [Zayas, 1987a] et Condor [Litzkow et Solomo, 1992] sont des exemples de systèmes qui fournissent respectivement un accès transparent aux canaux de communication et aux fichiers (par exemple, l'emplacement/distribution est caché). Les mêmes fonctionnalités pourraient être mises en œuvre par un intergiciel basé sur Java où la mobilité et la persistance de processus bénéficieraient de l'accès transparent aux objets d'entrées/sorties.

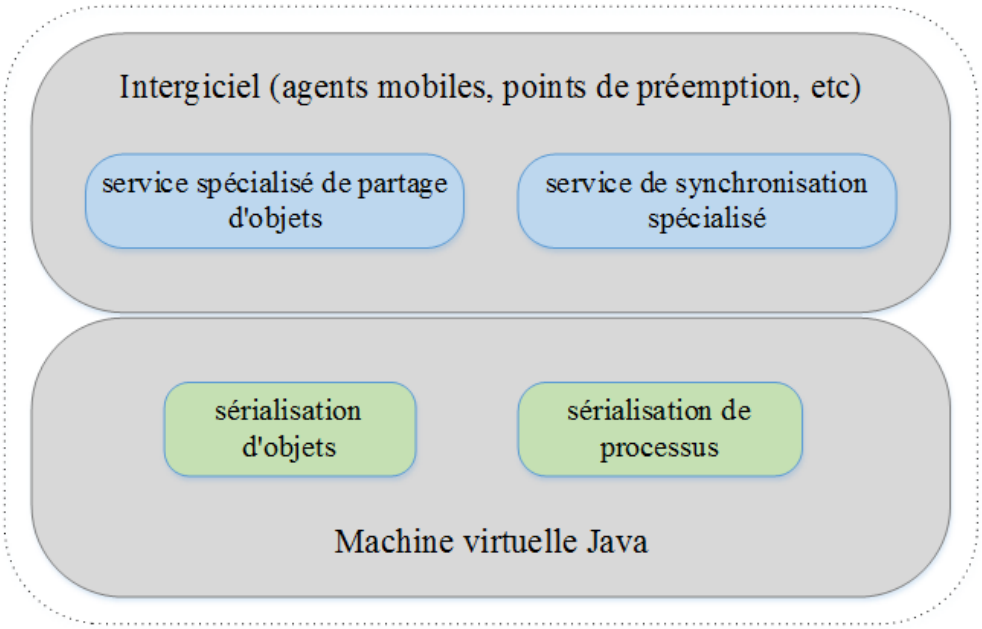


Figure 2.25 Sérialisation de processus, un élément de base dans un environnement d'intergiciel

La principale difficulté lors de la capture de l'état d'exécution d'un processus est son inaccessibilité aux programmes Java. Afin de remédier à ce problème, deux approches principales peuvent être suivies [Bouchenak et al., 2004] [Bouchenak, 2001], à savoir:

- L'approche du niveau noyau (machine virtuelle Java);
- L'approche du niveau application.

L'approche la plus intuitive pour accéder à l'état d'un processus Java consiste à ajouter de nouvelles fonctions à l'environnement Java pour exporter l'état des *processus* de la JVM. Les systèmes tels que Sumatra [Acharya et al., 1996], Merpati [Suezawa, 2000], ITS [Bouchenak et Hagimont, 2000] et CIA [Illmann et al., 2001] utilisent l'approche de l'extension de la machine virtuelle Java. Cette approche permet l'accès complet à tout l'état d'un processus Java mais son principal inconvénient réside dans sa dépendance à une extension particulière de la JVM. En effet, le mécanisme de sérialisation de processus ne peut donc pas être utilisé sur des machines virtuelles existantes.

Pour remédier au problème de la non-portabilité du mécanisme de sérialisation de processus dans des différents environnements de la machines Java, certains projets proposent une approche au niveau de l'application sans avoir recours à une extension de la JVM. Dans cette approche, le code de l'application est transformé par un préprocesseur avant l'exécution afin d'attacher un objet de sauvegarde du programme exécuté par le *processus* Java et d'ajouter de nouveaux états dans ce programme. Les codes ajoutés gèrent la capture de l'état du *processus* et les opérations de restauration. Ils stockent les informations d'état dans un objet de sauvegarde qui peut ainsi être sérialisé. Plusieurs systèmes de migration de *processus* Java suivent cette approche: Wasp [Funfrocken, 1998] et JavaGo [Sekiguchi, 1999] fournissent un préprocesseur de code source Java alors que Brakes [Truyen, 2000] et JavaGoX [Sakamoto et al., 2000] s'appuient sur un préprocesseur de bytecode.

Le principal avantage de la mise en œuvre du niveau application est la portabilité des mécanismes à tous les environnements Java. Cependant, ces mécanismes ne sont pas en mesure d'accéder à l'état d'exécution complète d'un processus Java. Dans le cas du système Wasp, par exemple, l'approche proposée ne permet pas de capturer les valeurs des résultats

intermédiaires [Bouchenak, 2001]. Considérons le programme présenté à la figure 2.26 pour illustrer une situation qui mène un résultat intermédiaire. À l'arrivée du programme sur la machine destinataire, la méthode *m2()* poursuit son exécution, puis se termine et retourne à la méthode *m1()* pour effectuer l'opération d'addition mais la valeur du premier opérande, la valeur 7, n'a pas été sauvegardée. Il en résulte une migration incomplète.

```
void m1 ( ) {
int resultat;
…
resultat = 7 + m2 ( ) ;
…
}
```

```
int m2 ( ) {
…
move ( ) ;
…
}
```

Figure 2.26 Exemple d'un code écrit en C pour illustrer une situation qui mène à un résultat intermédiaire [Bouchenak, 2001]

Bytecode

Le bytecode Java fournit un ensemble d'instruction qui est très similaire à celui d'un processeur matériel. Chaque instruction spécifie l'opération à exécuter, le nombre d'opérandes et les types d'opérandes manipulés par l'instruction. Par exemple, les instructions *iadd*, *ladd*, *fadd* et *dadd* s'appliquent respectivement sur deux opérandes de type *int*, *long*, *float, double* et retournent un résultat du même type.

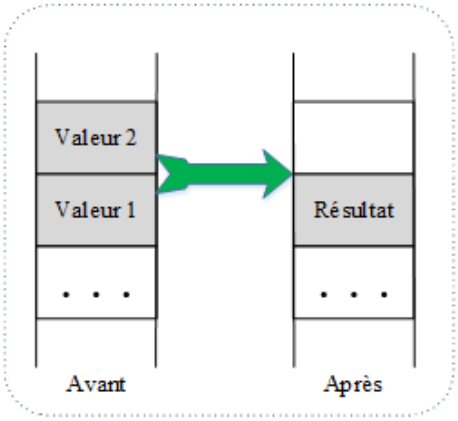


Figure 2.27 Addition de deux nombres entiers dans la machine virtuelle Java

L'exécution de *bytecode* dans la JVM est basée sur une pile appelée la pile d'opérandes. La figure 2.27 illustre l'exécution de l'instruction *iadd* qui additionne deux nombres entiers. Avant l'appel de l'instruction iadd, les deux nombres entiers sont empilés sur la pile et une fois que l'opération est terminée, le résultat est déposé au-dessus de la pile.

Moteur d'exécution

La première génération de la JVM était basée sur un interprète Java qui permettait de traduire chaque instruction de *bytecode* en code natif (le code binaire, c'est-à-dire le code machine exécuté par le processeur sous-jacent). Afin d'améliorer les performances, la deuxième génération de la JVM a intégré à Java une compilation à la volée (*Just-In-Time, JIT*), un compilateur qui compile de façon dynamique, c'est-à-dire des méthodes Java, bytecode, en code natif [Popovici et al., 2003] [Suganuma et al., 2000]. Ainsi, les appels ultérieurs et les exécutions de ces méthodes ne sont plus basés sur l'interprète Java, ils s'appuient directement sur le processeur sous-jacent et, par conséquent, s'exécutent beaucoup plus rapidement.

Structure de l'état d'exécution d'un processus Java

La spécification de la machine virtuelle Java définit plusieurs zones de données d'exécution [Lindholm et Yellin, 1999]. Nous nous intéressons particulièrement aux zones de données décrivant l'état d'exécution du processus Java, comme nous le montre la figure 2.28 [Bouchenak et al., 2004] [Bouchenak, 2001]. Cela comprend:

- Une *pile Java* associée à chaque processus qui consiste à une succession de trames. Une nouvelle trame est empilée sur la pile à chaque fois qu'une méthode Java est appelée par le processus et dépilée de la pile lors du retour de la méthode. Une trame comporte une table contenant les variables locales de la méthode associée et une pile qui contient les résultats intermédiaires de l'opération. Les valeurs des variables locales et des opérandes peuvent être de différents types: entier, flottant, référence Java, etc. Une trame contient également des registres comme le compteur ordinal (*PC, Program Counter*) et le sommet de la pile.
- Un *tas d'objet* de la JVM qui inclut tous les objets créés pendant la durée de vie de la machine virtuelle Java et un tas associé à chaque fil d'exécution qui se compose de

tous les objets utilisés par le *processus* (objets accessibles à partir de la pile du *processus* Java).

- Une *zone de méthode* de la JVM qui inclut toutes les classes qui ont été chargées par la JVM et une zone de méthode associée à chaque fil d'exécution qui contient les classes utilisées par le fil (des classes où certaines méthodes sont faites référence par la pile du processus).

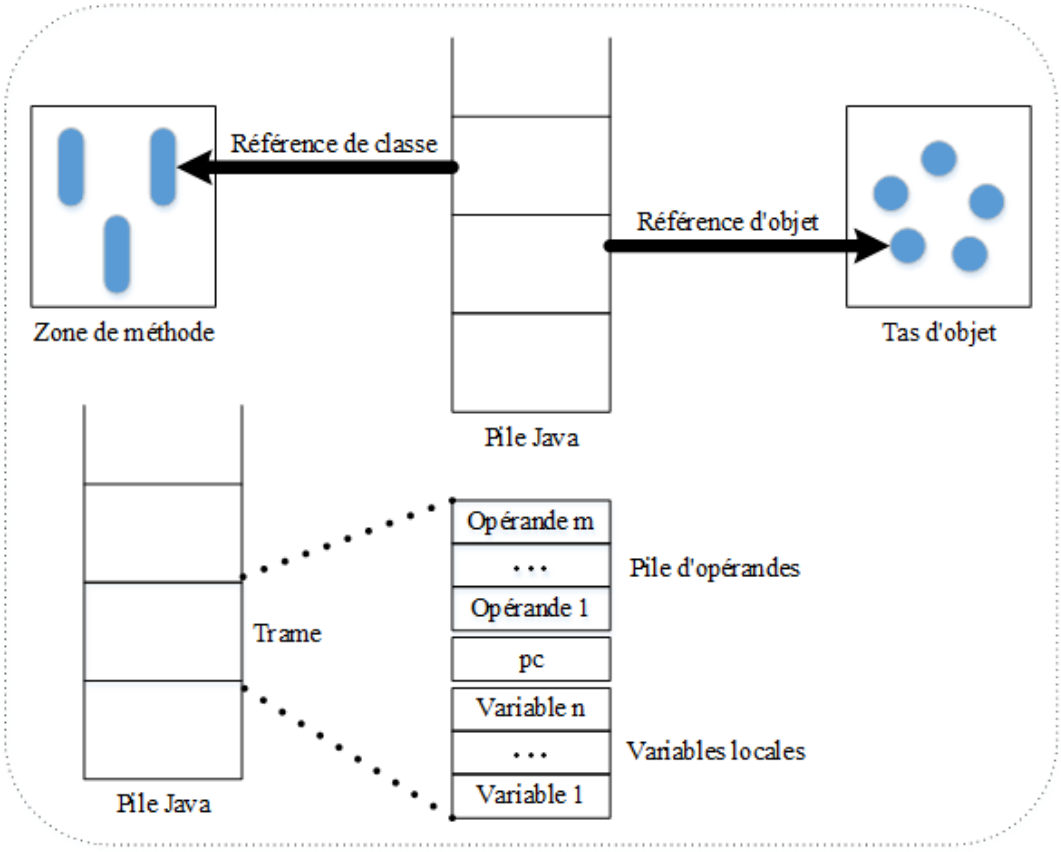


Figure 2.28 État de fil d'exécution Java [Bouchenak et al., 2004]

Les principales difficultés lors de l'extension de la JVM afin de réaliser la sérialisation d'un processus sont les suivantes:

a) L'accès à l'état d'exécution des processus. Cela est nécessaire pour construire un mécanisme de sérialisation de processus Java.

b) La mise en œuvre d'un état des processus Java portable. Cela est nécessaire pour suivre l'esprit de la portabilité Java (portabilité du code, portabilité des données et dans ce cas la portabilité du fil d'exécution).

c) La mise en œuvre d'un mécanisme de sérialisation de processus compatible avec le compilateur Java à la volée (le compilateur JIT). Cela est nécessaire afin de fournir une approche efficace pour les environnements Java.
d) La proposition d'un service générique de sérialisation. Pour les fils d'exécution, cette approche a été suivie afin de respecter la généricité et la réutilisabilité nécessaires.

Les approches proposées pour surmonter les difficultés lors de l'extension de la JVM afin de réaliser la sérialisation d'un processus sont les suivantes [Bouchenak et al., 2004]:

a) L'extension de la JVM afin d'externaliser l'état du fil et la mise à disposition d'une nouvelle API qui permet au programmeur d'accéder à cet état.
b) La mise en œuvre d'un mécanisme d'inférence de types qui transforme une structure de données non-portable d'un état de fil à une structure de données portable. Dans une JVM qui ne fournit pas d'information de type (par exemple, la norme JVM), l'utilisation de l'inférence de type est la seule technique qui résout le problème de la portabilité sans aucune baisse des performances.
c) L'utilisation d'une technique de dés-optimisation dynamique de code compilé à la volée (la compilation JIT). L'utilisation dés-optimisation dynamique est la seule technique qui permet de revenir aux méthodes compilées à la volée afin de fournir un mécanisme de sérialisation de processus Java qui est compatible avec la compilation à la volée. En effet, cette technique permet de revenir, au cours de l'exécution d'un code compilé, à la version interprétée de ce code. Elle est utilisée par exemple lorsqu'il faut «défaire» la compilation afin de faire une expansion des méthodes pour prendre en compte un nouveau code Java chargé dynamiquement.
d) Une conception générique qui suit l'approche orientée objet et la hiérarchie des classes afin de proposer un mécanisme de fil générique, adaptable et une sérialisation réutilisable.

État de processus non accessible

L'état de processus Java (la pile Java, le tas, la zone de méthode) est interne au JVM. Autrement dit, il n'y a pas d'API Java standard qui permet au programmeur d'accéder à la pile Java d'un processus (fil d'exécution), au tas (les objets utilisés par un fil) ou à la zone des

méthodes (les classes utilisées par un fil). Cet état ne peut donc pas être directement capturé pour mettre en œuvre la sérialisation de fil d'exécution. Pour remédier à ce problème, la JVM a été étendue pour pouvoir, d'une part, externaliser l'état de fil d'exécution Java et d'initialiser un fil avec un état particulier (de-sérialisation de processus), d'autre part.

État de processus non portable

Contrairement au tas et à la zone de méthode qui comprennent des informations portables sur des architectures hétérogènes (l'objet Java et le bytecode respectivement), la pile Java est mise en œuvre dans la plupart des machines virtuelles Java en tant que structure de données native (structure C). Par conséquent, la représentation des informations contenues dans la pile Java dépend de l'architecture sous-jacente. Afin de sérialiser un fil pour qu'il soit portable sur des plateformes hétérogènes, le mécanisme de sérialisation de fil doit traduire la structure de données non-portable représentant un état (la structure C) dans une structure de données portable (l'objet Java). Le mécanisme de dé-sérialisation du fil doit effectuer le travail inverse afin de restituer le processus.

La traduction de la pile Java dans une structure de données portable consiste, plus précisément, à traduire les valeurs natives des variables locales et les opérandes vers des valeurs Java. Cette traduction nécessite la connaissance des types de valeurs. Cependant, la pile Java ne fournit aucune information sur les types des valeurs qu'elle contient: un mot de quatre octets peut représenter une référence Java ainsi qu'une valeur de type *entier* ou *flottant*. Par conséquent, dans le cas présent, le problème reste à savoir comment déduire les types des données stockées dans la pile Java?

Le seul endroit où ces types sont connus est dans le *bytecode* des méthodes qui empilent les données sur la pile. L'approche la plus intuitive reste ainsi la modification de l'interprète Java pour qu'à chaque fois qu'une instruction de *bytecode* empile une valeur sur la pile, le type de cette valeur soit stocké «quelque part» (c'est-à-dire sur une pile de types associée au fil).

Il existe deux approches de construction de la pile de types associée à un processus Java [Bouchenak, 2001]: une construction de la pile à l'interprétation du *bytecode* et une construction de la pile au moment de la capture de l'état d'exécution d'un processus.

Construction de la pile à l'interprétation du bytecode

Dans cette approche, une pile de types est associée à un processus au moment de sa création. Comme la pile Java du processus, la pile de types évolue avec le processus, au fur et à mesure que celui-ci interprète les instructions de bytecode [Bouchenak, 2001] [Bouchenak, 2000].

La figure 2.29 nous présente un programme Java et son bytecode. Ce programme affecte la valeur entière 5 à la première variable locale de la méthode m. Le processus Java, qui exécute ce programme, a une pile Java et une pile de types qui lui sont associées comme illustrées à la figure 2.30.

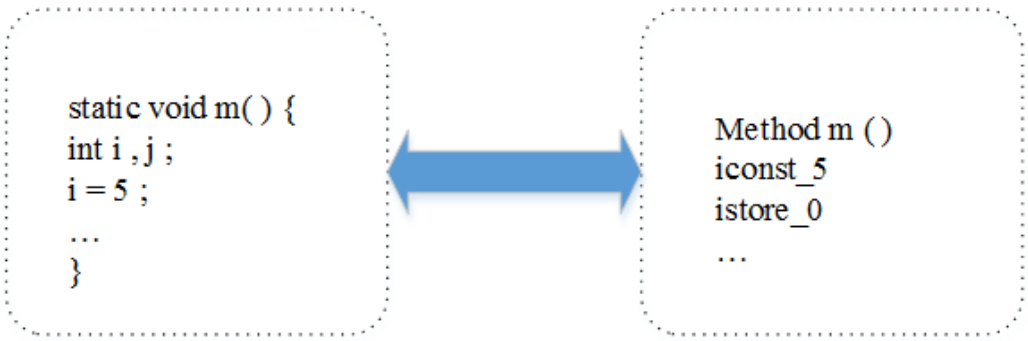


Figure 2.29 Programme Java et son Bytecode

Comme nous le montre la figure 2.30 (a), la pile Java contient une trame Java qui représente l'état d'exécution de la méthode *m*. Avant l'interprétation de l'instruction *istore_0*, la trame a deux variables locales non-initialisées et un résultat intermédiaire de valeur entière 5 (précédemment empilé par l'instruction *iconst_5*). L'interprétation de l'instruction *istore_0* dépile la valeur entière en sommet de pile d'opérandes et la stocke dans la variable locale d'indice 0. L'interprétation de cette instruction de *bytecode* permet donc de connaître le type de la première variable locale: le type *int*.

Comme nous le montre la figure 2.30 (b), la pile de types du processus contient une trame de types associés à la méthode *m*. Avant l'interprétation de l'instruction *istore_0*, la trame a deux variables locales de types inconnus et un type *int* sur la pile d'opérandes (précédemment reconnu grâce à l'instruction *iconst_5*). L'interprétation de l'instruction *istore_0* dépile le type *int* en sommet de pile d'opérandes et le stocke dans la variable locale d'indice 0: cette variable a pour type *int*.

Un premier prototype de sérialisation de processus Java suit cette approche, il est appelé *ITS (Interpreter-based Thread Serialization)* [Bouchenak et Hagimont, 2000]. L'inconvénient de cette approche est qu'elle introduit un surcoût significatif sur les performances puisque la pile de types évolue en parallèle à la pile Java du processus avec l'interprétation de *bytecode*.

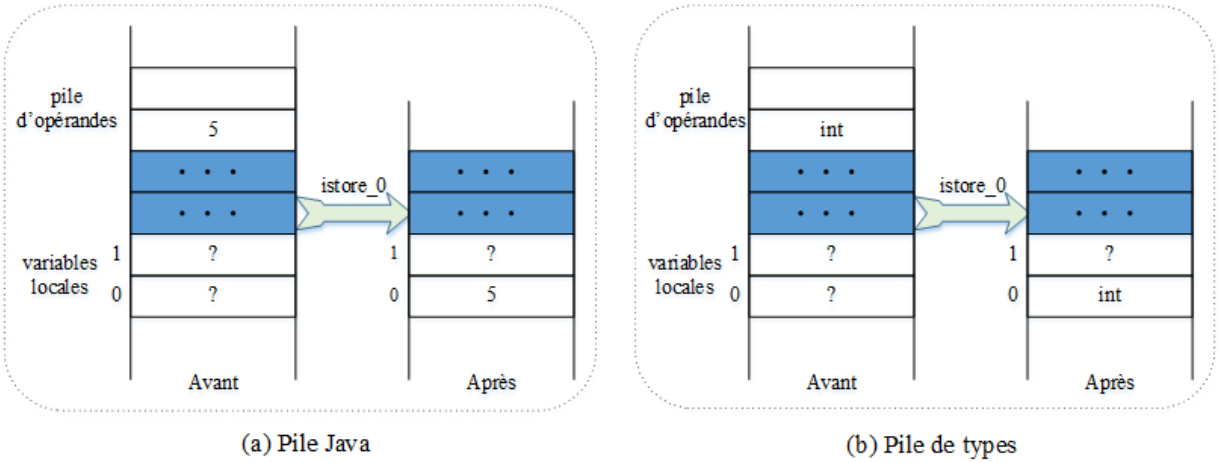


Figure 2.30 Interprétation de bytecode/reconnaissance de types

Construction de la pile de types à la capture

Afin d'éviter toute surcharge, l'inférence de type ne doit pas être effectuée pendant l'exécution du processus mais seulement au moment de sérialisation de processus. Ainsi, une approche, dans laquelle le bytecode exécuté par le processus analysé au moment de sérialisation de processus, a été proposée [Bouchenak et al., 2004]. Avec cette analyse, les types de données empilés sont récupérés et utilisés pour construire la structure de données portables qui représente la pile du processus Java. Ainsi, grâce à cette approche, l'interprète Java reste inchangé et aucune baisse des performances n'est observée lors de la sérialisation de fil

d'exécution. Cette approche est appelée CTS (Capture time-based Thread Serialization) [Bouchenak, 2002] [Bouchenak, 2001].

Pour éviter un surcoût lors de la sérialisation, les deux principes suivants ont été suivis:

- Aucun traitement supplémentaire n'est exécuté en parallèle avec l'interprétation de *bytecode*: Tout est fait au moment de la sérialisation. Ceci est réalisé en utilisant une technique d'*inférence de types* (voir la description ci-dessous) appliquée au moment de sérialisation.
- La compatibilité de sérialisation de processus Java avec des techniques actuelles de compilation à la volée. Le problème ici est d'être en mesure d'effectuer de sérialisation de processus, même si la pile du processus Java ne reflète pas vraiment l'état d'exécution en cours. C'est le cas lorsque des méthodes Java exécutées par le processus sont compilés à la volée (c'est-à-dire, leur exécution est basée sur la pile des processus natifs et non sur la pile Java). Pour faire face à ce problème, il a été proposé d'utiliser une *technique de dés-optimisation dynamique du code Java compilé* (voir la description ci-dessous).

Mise en œuvre de la capture/restauration

Comme décrit précédemment, les principales difficultés lors de l'extension de la JVM afin de réaliser la sérialisation de processus et les approches pour les surmonter consistent à:

- Accéder un état d'exécution Java: la JVM est étendue;
- Fournir l'état d'un processus portable: une technique d'inférence de types est proposée;
- Construire une sérialisation de processus sans surcoût: une inférence de types en combinaison avec une technique de dés-optimisation dynamique du code Java compilé est utilisée;
- Adapter la sérialisation de processus: une conception générique du mécanisme de sérialisation est employée.

Inférence de types

Le mécanisme d'inférence de types proposé vise à construire une pile de types qui reflète les types des valeurs (les variables locales et les opérandes) contenu dans la pile du processus Java. Comme la pile Java, la pile de types se compose d'une succession de trames qu'on appelle des trames de types (voir la figure 2.31). Une trame de types sur une pile de types est associée à chaque trame sur la pile Java. Une trame de types contient deux structures de données principales: une table qui décrit les types des variables locales de la méthode associée et une pile de types d'opérandes qui donne les types des résultats intermédiaires de la méthode. La pile de types d'un fil est construite comme suit. Au moment de la sérialisation de processus, une pile de types vide est initialement associée à la pile Java du fil d'exécution. Pour chaque trame sur la pile Java, une trame de type vide est initialement empilée sur la pile de types associée. Les types des variables locales et les opérandes de la trame Java sont alors déduits comme suit. Le *bytecode* de la méthode associée fait l'analyse syntaxique depuis le début de la méthode jusqu'au point de sortie de la méthode (le PC de la trame Java donne le point de sortie et représente la dernière instruction exécutée dans la méthode). Suivant ce chemin de code, les instructions de *bytecode* sont analysées et les types des valeurs qu'ils manipulent sont déduits et stockés dans la trame de types, comme des variables locales ou des types d'opérandes.

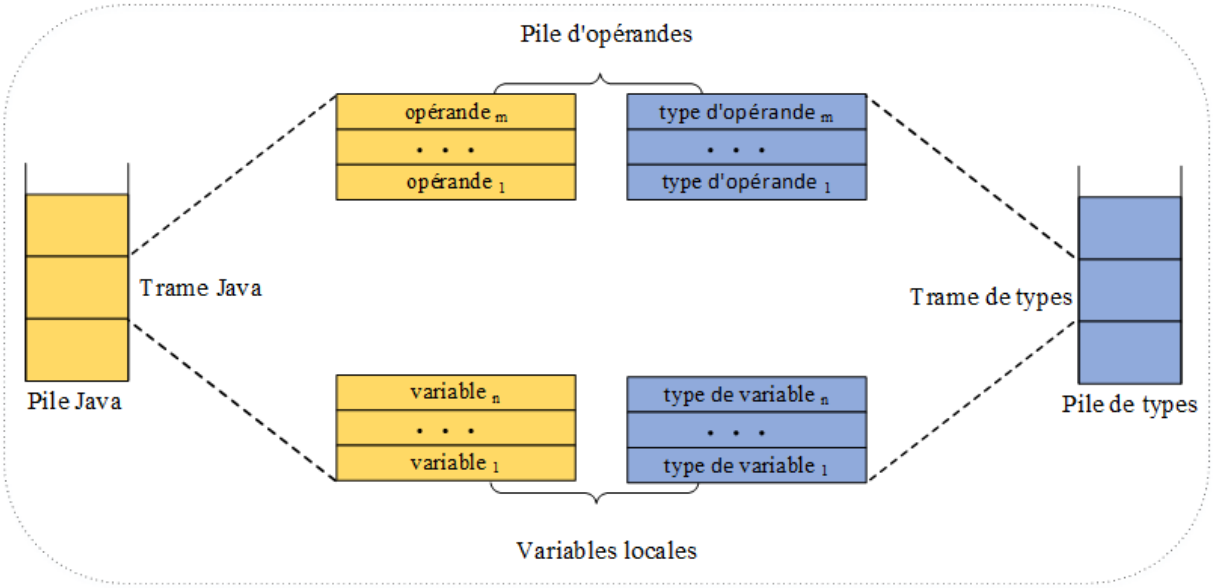


Figure 2.31 Pile de types versus pile Java

Cependant, une question se pose sur le chemin à suivre pour l'inférence de type lors de déduction des types quand il existe plusieurs chemins entre le début du code de la méthode et le point de sortie de la méthode. Il est important de noter ici que des chemins de code différents peuvent supposer différents types pour un même objet (variable locale ou opérande) sur la pile Java. Ce problème est illustré par l'exemple d'une méthode *m* représenté par un code source Java, son *bytecode* équivalent et un graphe de ses flots d'exécution associé (voir la figure 2.32).

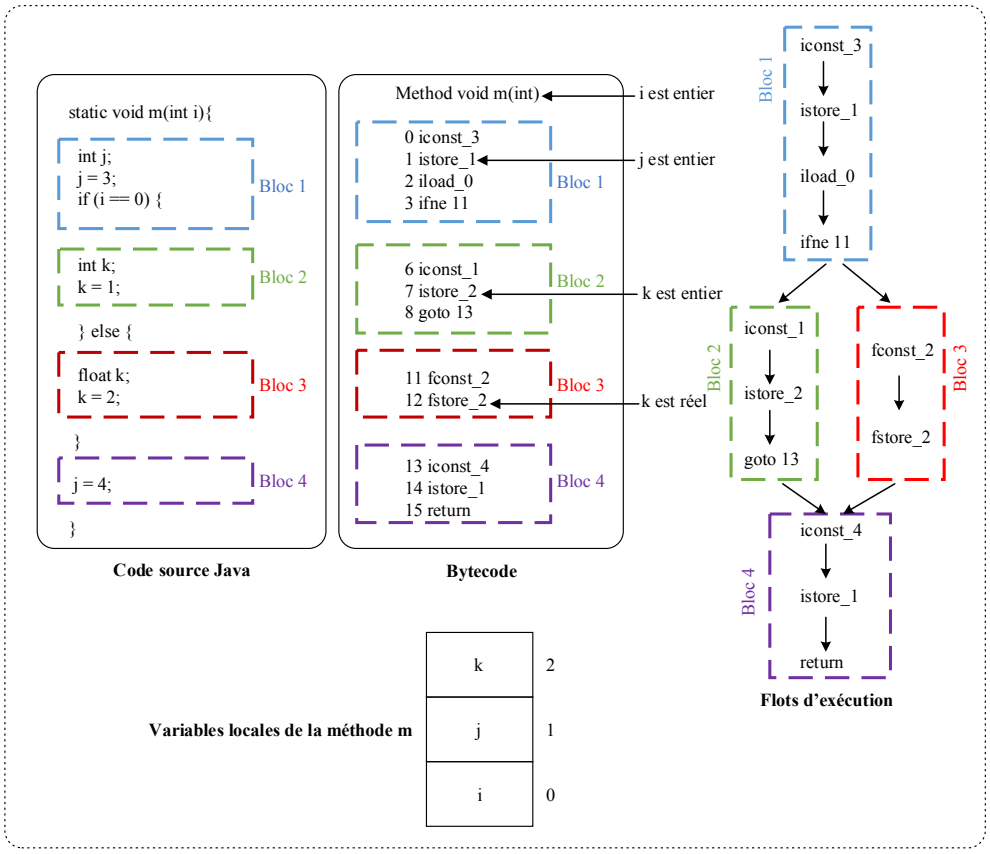


Figure 2.32 Code source Java, son bytecode et ses flots d'exécution

Dans ce programme, les variables locales *i* et *j* sont déclarées dans le bloc 1 et représentent des valeurs de type entier et la variable locale *k* représente une valeur de type entier dans le bloc 2 et de type *flottant* dans le bloc 3. Cette variable est mise en œuvre par la même entrée dans la table des variables locales de la trame Java (la variable à l'indice 2, manipulée à des lignes 7 et 12 dans le *bytecode*).

Lors de la sérialisation du processus qui exécute la méthode *m* et compte tenu de la trame Java non typé et son b*ytecode,* comment sont déterminés les types des variables locales et des opérandes de la méthode? Quatre cas sont possibles:

1. Le point de sortie (valeur de PC) est dans le *bloc 1*. Dans ce cas, il n'y a qu'un seul chemin possible depuis le début du code jusqu'au point de sortie. L'analyse de ce chemin détermine que la variable locale *i* est une valeur entière grâce à la signature de la méthode et la variable locale *j* qui est de type entier grâce à l'instruction *istore_1* à la ligne 1 du *bytecode*.
2. Si le point de sortie est dans le *bloc 2*, alors le seul chemin pour y arriver est le *bloc 1* à *bloc 2*. Lorsqu'on analyse ce chemin, les variables locales *i* et *j* sont reconnus comme étant des types entiers (comme dans le premier cas) et la variable locale *k* est de type entier grâce à l'instruction *istore_2* à la ligne 7 du *bytecode*.
3. Dans le cas où le point de sortie est dans *bloc 3*, il n'y a qu'un seul chemin pour y arriver: bloc1 à bloc3. Ce cas est similaire à la seconde, la seule différence est que l'analyse du chemin détermine la variable *k* comme étant un type *flottant* grâce à l'instruction *fstore_2* à la ligne 12 du *bytecode*.
4. Enfin, si le point de sortie est dans le *bloc 4*, il y a deux chemins possibles: soit *bloc 1 – bloc 2 – bloc 4* ou *bloc 1 – bloc 3 – bloc 4*. Dans ce cas, quel chemin le code doit-il suivre pour l'inférence de type? Est-ce que la variable *k* est de type entier ou de type *flottant*?

La solution à ce problème est basée sur deux propriétés invariantes du *bytecode* [Bouchenak et al., 2004] [Engel, 1999]: À n'importe quel point du programme et quel que soit le chemin suivi pour atteindre ce point:

1. Propriété 1: Les piles d'opérandes construites en suivant chaque chemin contiennent des valeurs de mêmes types.
2. Propriété 2: Les variables locales construites en suivant chaque chemin sont de mêmes types ou inutilisées si leurs types diffèrent.

En fait, les deux propriétés invariantes assurent la correction du bytecode [Bouchenak et al., 2004] [Bouchenak, 2001] [Engel, 1999]. D'après ces deux propriétés, arrivé au début du *bloc 4* et quel que soit le chemin suivi à l'exécution de la méthode [Bouchenak et al., 2004]:

- La pile d'opérandes de trame Java de la méthode est vide (avant l'exécution de *j = 4* par le code source Java).
- Les variables locales *i* et *j* sont reconnues comme étant des entiers parce qu'elles ont préalablement été utilisées et la variable locale *k* est inutilisée car même si son type diffère dans les blocs 2 et 3, elle reste interne à ces deux blocs et inconnue du bloc 4.

Dans cette étude [Bouchenak et al., 2004], les auteurs ont mis en œuvre un algorithme qui infère les types des valeurs (les variables locales et les opérandes) sur une trame Java, et plus généralement sur la pile d'un processus Java, en un seul passage du *bytecode*. Cet algorithme est appliqué au moment de sérialisation de processus et permet de:

- déterminer, pour chaque code de la méthode en cours d'exécution par le fil, un chemin de code à partir du début du code de la méthode jusqu'au point de sortie de la méthode (la valeur de PC) et
- déduire les types de données manipulées par les instructions de *bytecode* contenues dans ce chemin.

Enfin, l'algorithme d'inférence construit une pile de types qui reflète les types des valeurs sur la pile du processus Java. Les informations de types résultantes sont ensuite utilisées afin de capturer la pile du processus Java dans un format portable. En outre, l'inférence de types est une technique utilisée dans la vérification *bytecode*, qui est généralement appliquée à Java au moment du chargement de classe [Lindholm et Yellin, 1999].

Technique de dés-optimisation dynamique du code Java compilé

La technique d'inférence de type décrite dans la section précédente exige l'accès à la pile Java du fil d'exécution. Toutefois, la pile Java ne reflète pas toujours l'état d'exécution actuel du processus puisque l'exécution de méthodes compilées à la volée n'est pas basée sur la pile du processus Java mais sur la pile native. La problématique ici est de permettre la sérialisation de processus même en présence de compilation à la volée.

Dans le cas présent, le mécanisme de sérialisation de processus aurait besoin des fonctions qui lui permettraient de restaurer les trames Java à partir des trames natives produites par le compilateur à la volée. La technique d'inférence de type pourrait ensuite être appliquée.

La machine virtuelle *HotSpot* de *Sun Microsystems* comprend un mécanisme qui assure une technique de dés-optimisation dynamique du code Java compilé. Ce mécanisme transforme les trames compilées à la volée en trames Java [Meloan, 1999]. La technique de dés-optimisation dynamique du code Java compilé a d'abord été utilisée dans le système de l'auto-débogage au niveau source, elle protège le débogueur des optimisations effectuées par le compilateur de dés-optimisation dynamique de code à la demande [Holzle et al., 1992)]. Ceci permet de déboguer son programme au niveau du code source même en présence d'optimisations de compilation.

Dans la machine virtuelle *HotSpot*, la technique de dés-optimisation dynamique du code Java compilé a été mise en œuvre pour traiter l'incohérence qui apparait lors de l'expansion de méthode compilée à la volée et le chargement dynamique de classes. La figure 2.33 illustre ce problème à l'aide d'un exemple où une méthode *m1* appelle une méthode *m2* d'une classe *C1*. Pour des raisons d'optimisation, le compilateur à la volée peut faire l'expansion de *m2* dans *m1*. Cependant, cette expansion peut devenir invalide si *C2*, une sous-classe de *C1* qui substitue *m2*, est dynamiquement chargée et si *getInstanceOfC1* appelée dans la méthode *m1* retourne une instance de *C2*. À partir d'une incohérence engendrée par l'optimisation (compilation et expansion), la technique dynamique de l'optimisation est utilisée pour restaurer un code valide interprété.

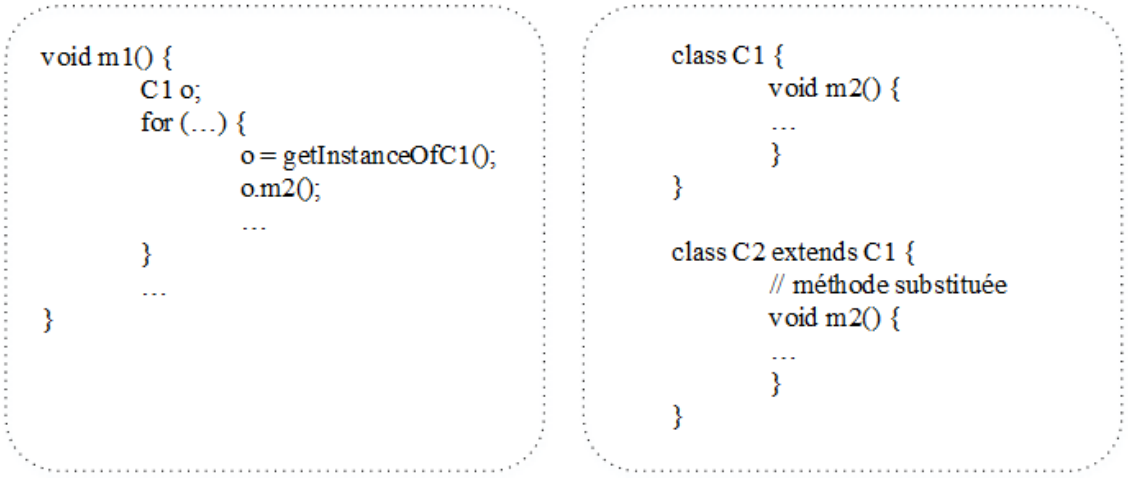


Figure 2.33 Expansion de méthode et le chargement dynamique de classes [Bouchenak et al., 2004]

Pour récapituler, la technique de dés-optimisation dynamique du code Java compilé est utilisée dans le domaine du débogage et du chargement de classes dynamiques. Dans le domaine de la sérialisation de processus, elle est utilisée comme suit. Au moment de la sérialisation, au cas où certaines méthodes Java aient été compilées à la volée, la technique de dés-optimisation dynamique est appelée à des trames de processus compilé à la volée. Ceci mène à la récupération de trames Java qui auraient été produites par l'interprète Java. Ainsi, l'algorithme d'inférence de types décrit précédemment peut être appliqué à ces trames Java et le processus peut être sérialisé.

Dans [Bouchenak et al., 2004] [Bouchenak, 2001], les auteurs proposent des mécanismes de capture de l'état d'exécution des processus qui répondent à deux exigences de Java:

1. *Les performances*: Les mécanismes permettent aux applications Java de bénéficier des techniques d'optimisation de l'exécution (compilation à la volée) proposées par la JVM.
2. *La portabilité*: les mécanismes prennent en compte la totalité de l'état d'exécution des processus Java, quelle que soit la nature du sous-système d'exécution sous-jacent. L'état capturé est ainsi un état portable qui reflète l'état d'exécution des méthodes Java interprétées et l'état des méthodes compilées à la volée.

Pour garantir les deux conditions précédentes, une technique de dés-optimisation, utilisée dans le domaine de débogage et de chargement de classes dynamiques, est employée.

Évaluation comparée des performances

Dans cette section, nous présentons les résultats de l'évaluation comparée des performances de certains systèmes de sérialisation de processus Java. Dans cette évaluation, deux mesures ont retenu notre attention:

1. Le surcoût sur les performances lors de l'exécution de code se définit comme la différence entre le temps nécessaire pour exécuter le code d'application sur un système qui fournit la sérialisation de processus et le temps nécessaire pour exécuter le même code sur un système qui ne fournit pas de sérialisation de processus.
2. La latence de sérialisation se définit, quant à elle, comme la somme du temps nécessaire pour la sérialisation/désérialisation de processus.

Pour évaluer la variation de coûts des performances sur un processus selon le nombre d'opérations qu'il exécute, un code Java basé sur l'algorithme récursif de *Fibonacci* est utilisé. Les tests de performances ont été réalisés dans les systèmes *JavaGo, Brakes, JavaGoX, ITS* et *CTS*. Rappelons que:

- *JavaGo* est un système de niveau application basé sur un préprocesseur du code source exécuté par un processus Java sérialisé;
- *Brakes* et *JavaGoX* sont des systèmes de niveau application basés sur un préprocesseur du bytecode exécuté par le processus Java sérialisé;
- *ITS* est un système de niveau JVM, c'est la première version développée par [Bouchenak et al., 2004]. En effet, c'est une extension de l'interprète Java;
- *CTS* est la version finale de niveau JVM proposée par [Bouchenak et al., 2004]. Elle est basée sur des techniques de l'inférence de types et de dés-optimisation dynamique du code Java compilé intégrés dans la JVM.

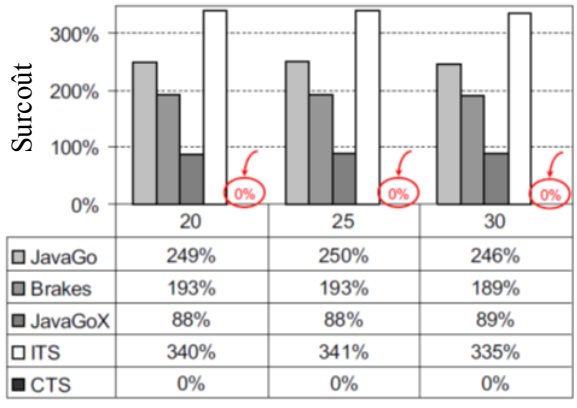
Le programme Fibonacci utilise les valeurs 20, 25 et 30 comme paramètres. La figure 2.34 (a) nous présente la variation de surcoûts sur l'exécution du programme *Fibonacci* selon que la compilation JIT est activée ou désactivée.

- *JavaGo*, *Brakes* et *JavaGoX* encourent des surcoûts de performance non négligeable (+88% à +250%) en raison du code inséré par le préprocesseur dans l'application. Dans cette évaluation, le système *JavaGo* a le deuxième plus haut surcoût parce qu'il

ajoute le code Java au code source de l'application. Brakes et JavaGoX se classent en troisième et quatrième place respectivement parce qu'ils ajoutent leur code au niveau de *bytecode*.

- *ITS* impose un surcoût important (+335% à 340%) en raison du traitement supplémentaire exécuté par l'interprète Java étendu sous-jacent à presque chaque interprétation *bytecode*.
- *CTS* n'encourt aucun surcoût car il n'impose pas de calcul supplémentaire.

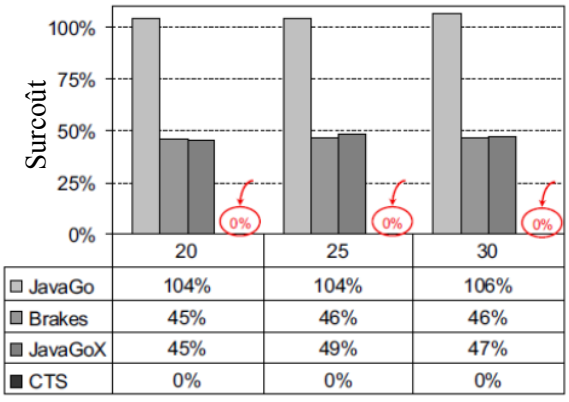
Dans la figure 2.34 (a), la performance illustrée représente les résultats du programme *Fibonacci* fonctionnant sans la compilation Java JIT. Ceci était nécessaire pour pouvoir comparer le système *ITS* avec d'autres systèmes. En effet, *ITS* est basé sur un interprète Java étendu et peut donc seulement être utilisé dans le mode interprété (sans la compilation JIT). Les surcoûts de performance effectifs (avec la compilation JIT) encourus par les autres systèmes *(JavaGo, Brakes, JavaGoX, CTS)* sont présentés à la figure 2.34 (b). Enfin, même si la compilation JIT réduit les surcoûts encourus par les exécutions de *JavaGo, Brakes* et *JavaGoX*, elle ne les annule pas (+45% à +106%), ce qui pénalise lourdement la sérialisation de processus Java.



	20	25	30
JavaGo	249%	250%	246%
Brakes	193%	193%	189%
JavaGoX	88%	88%	89%
ITS	340%	341%	335%
CTS	0%	0%	0%

Paramètre de Fibonacci

(a) Compilation Java JIT est désactivée



	20	25	30
JavaGo	104%	104%	106%
Brakes	45%	46%	46%
JavaGoX	45%	49%	47%
CTS	0%	0%	0%

Paramètre de Fibonacci

(b) Compilation Java JIT est activée

Figure 2.34 Surcoûts sur l'exécution du programme *Fibonacci* [Bouchenak et al., 2004]

Les résultats de tests montrent que *CTS* est le seul système qui n'encourt pas un surcoût lors de la sérialisation de processus. Ce comportement n'est pas magique: il est dû au fait que, avec

CTS, tout traitement supplémentaire est réalisé au moment de sérialisation de processus. Dans les tests suivants, nous visons à discuter de la relation entre la variation:

- Les surcoûts lors de la sérialisation de processus;
- La latence d'un processus sérialisé.

Pour pouvoir comparer la latence des différents systèmes, un niveau intermédiaire des opérations de sérialisation a été construit afin d'homogénéiser l'environnement d'évaluation [Bouchenak et al., 2004].

La latence de sérialisation de processus varie en fonction de la taille de l'état du processus. Le programme de tests a été écrit de manière à ce que le nombre d'objets utilisés reste fixe afin de mettre l'accent sur la variation du nombre de trames sur la pile du processus Java pour des raisons de simplicités. La figure 2.35 présente les coûts de l'exécution versus la latence lorsque la pile Java du processus sérialisé contient cinq trames dans les systèmes suivants: *JavaGo, Brakes, JavaGoX, ITS* et *CTS*.

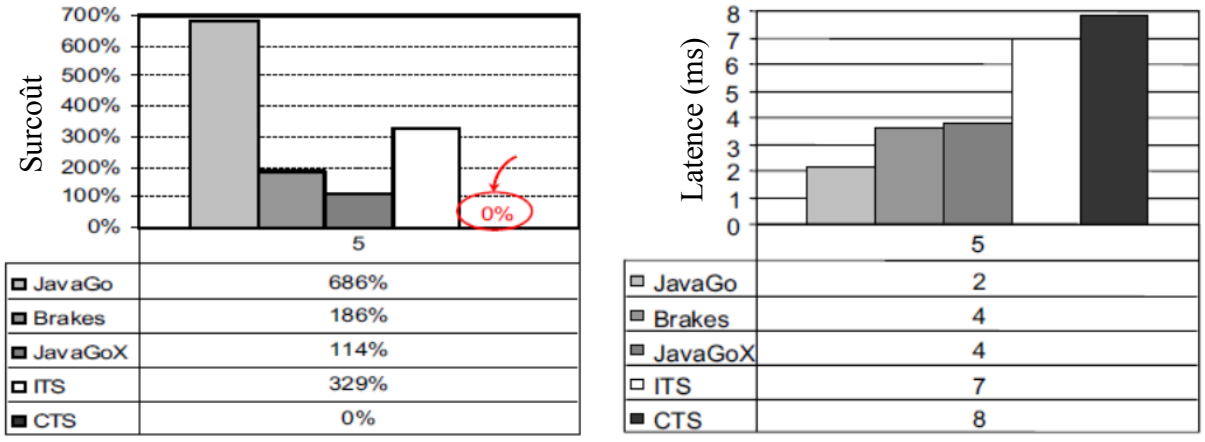


Figure 2.35 Surcoûts sur l'exécution versus la latence d'un processus sérialisé avec cinq trames (compilation Java JIT désactivée) [Bouchenak et al., 2004]

La figure 2.36 présente les coûts de l'exécution versus la latence lorsque la pile Java du processus sérialisé contient dix trames dans les systèmes suivants: *JavaGo, Brakes, JavaGoX, ITS* et *CTS*.

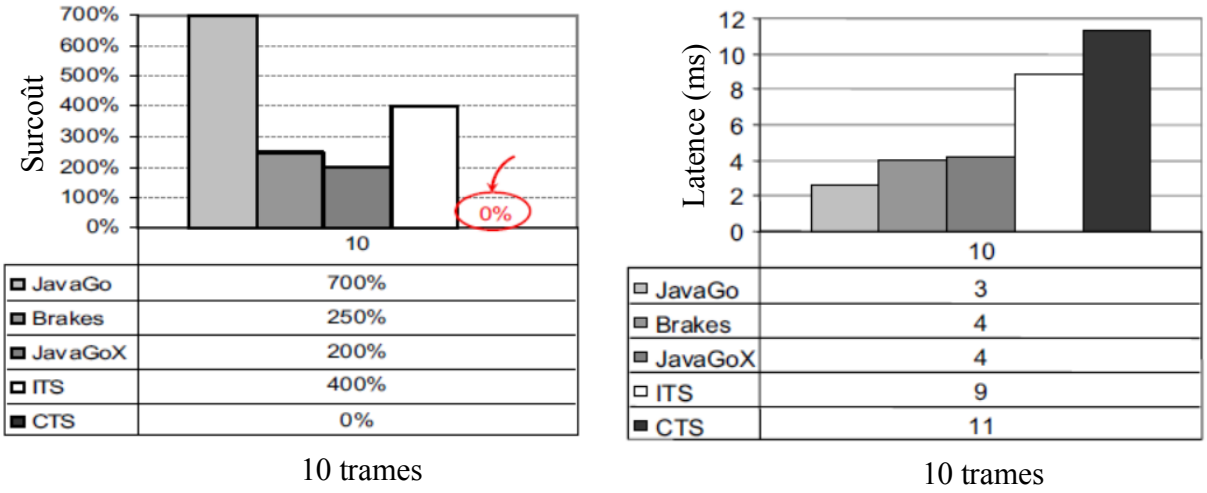


Figure 2.36 Surcoûts sur l'exécution versus la latence d'un processus sérialisé avec dix trames (compilation Java JIT désactivée) [Bouchenak et al., 2004]

Les deux figures montrent que:

- Pour les systèmes de niveau application *(JavaGo, Brakes, JavaGoX)*, le surcoût sur le processus sérialisé et la latence de sérialisation sont inversement proportionnels. Ceci s'explique par le fait que plus le traitement est effectué pendant l'exécution du processus (surcoût), moins il y a de traitement nécessaire au moment de la sérialisation. Nous remarquons aussi que les systèmes *Brakes* et *JavaGoX* présentent des comportements similaires en raison de leur approche de niveau *bytecode*.
- La performance de *ITS* se situe entre celle de niveau source *(JavaGo)* et celle de niveau bytecode *(Brakes, JavaGoX)*. Cependant, *ITS* présente une latence de sérialisation plus élevée comparée à *JavaGo, Brakes* et *JavaGoX*. Ceci est dû au fait que *ITS* capture l'état de processus complet et fait ainsi plus de traitement au moment de la sérialisation.
- *CTS* n'impose pas de surcoût de performance mais il présente la latence de sérialisation de processus la plus élevée car tout est fait au moment de la sérialisation. En outre, il est important de noter que cette mise en œuvre actuelle de *CTS* ne comprend pas la dés-optimisation dynamique du code Java compilé. Ainsi, la mise en œuvre qui intègre la dés-optimisation et la ré-optimisation dynamique aurait probablement une latence plus élevée.

Une remarque s'impose: le surcoût sur l'exécution est plus pénalisant que le coût sur la latence de migration car on ne migre pas fréquemment.

Les résultats de performance présentés ici ont été obtenus dans l'environnement suivant [Bouchenak et al., 2004]:

- Processeur Pentium III, 1 GHz monoprocesseur, 256 Mo de RAM;
- Windows NT avec SP 4;
- Environnement de développement Java/Version 1.2.2 de Sun Microsystems, également connu sous le nom Java 2 SDK/Version 1.2.2.

2.5 Conclusions

Dans ce chapitre, nous nous sommes intéressés particulièrement à la migration de processus. Nous avons commencé par définir les différents éléments qui permettent la migration de processus. La migration peut être mise en œuvre aux niveaux application ou noyau. Les mises en œuvre de migration au niveau application ne permettent pas d'accéder à l'état de noyau d'où leur incapacité à faire migrer tous les processus.
La migration de processus au niveau noyau implique des extensions du système d'exploitation sous-jacent. L'avantage de la migration au niveau noyau est qu'elle offre une mobilité de processus complète (donc forte) plus efficacement mais elle mène à une complexité supplémentaire. Un autre avantage de mobilité de processus au niveau noyau est la transparence à l'application.

Nous avons classé les travaux de migration des processus en deux grandes catégories: les systèmes homogènes et ceux hétérogènes. Dans la migration de processus sur les systèmes homogènes, nous avons exposé les notions fondamentales: les algorithmes de migration, les métriques pour l'évaluation de temps de migration, les plateformes, l'espace d'adressage, etc. Nous avons également présenté une évaluation qualitative et quantitative des algorithmes de migration de processus dans différents systèmes homogènes.

Dans les systèmes hétérogènes, nous avons tout d'abord défini le concept de migration de processus et établi les critères de performances. Puis, nous avons exposé l'approche de réflexivité qui est utilisée dans certains travaux de migration de processus dans les systèmes hétérogènes. Ensuite, nous avons présenté une synthèse des travaux de migration de processus dans des environnements hétérogènes. Nous avons décrit le projet Attardi [Attardi et al., 1988] qui utilise l'approche de la réflexivité pour la mise en œuvre de son système de migration de processus par interprétation ou traduction. Nous avons également présenté le système Tui [Smith et Hutchinson, 1997] qui supporte la migration de processus utilisant des codes natifs. Le système Tui utilise une version modifiée d'un compilateur qui rajoute des informations permettant la capture/restauration d'une pile d'exécution dans quatre architectures.

Nous avons également exposé de travaux de migration de processus sur la machine virtuelle Java. Dans ces travaux, la migration complète de processus est mise en œuvre via une extension de la machine virtuelle Java. Nous avons décrit comment la sérialisation d'objets (fournie déjà par machine virtuelle Java) peut être transformée en une sérialisation de processus portable. Pour garantir les performances et la portabilité dans la machine virtuelle Java, une technique de dés-optimisation, utilisée dans le domaine de débogage et de chargement de classes dynamiques, est employée.

CHAPITRE 3
AGENTS MOBILES ET LEURS SPÉCIFICITÉS

Nous venons de présenter, dans le chapitre précédent, la migration de processus dans des systèmes homogènes et hétérogènes ainsi que les mécanismes indispensables qui permettent leur mise en œuvre.

Dans ce chapitre, nous étudions les agents mobiles et leurs spécificités. Nous commençons par les définir en mettant l'accent sur leurs caractéristiques propres. Puis, nous présentons les travaux sur la modélisation des systèmes à base d'agents mobiles. Nous décrivons les efforts de standardisation des plateformes d'agents mobiles pour promouvoir leur interopérabilité. Nous exposons les plateformes supportant le modèle des agents mobiles et leurs domaines d'application. Nous présentons ensuite un bilan des avantages et inconvénients des agents mobiles. Nous finissons ce chapitre qui termine la revue des travaux dans le domaine de la mobilité des applications (la migration de processus et les agents mobile) par une conclusion et nous nous positionnons par rapport aux autres recherches.

3.1 Concept d'agents

La définition d'un agent suscite de nombreuses polémiques dans le milieu de la recherche scientifique et industrielle. En simplifiant au maximum, il y a d'un côté ceux qui considèrent les agents presque comme des êtres humains et de l'autre ceux qui les assimilent à de simples logiciels. Ainsi, Ferber [Ferber, 1999] définit un agent comme une entité physique ou virtuelle qui possède les propriétés suivantes:

- capable d'agir dans un environnement,
- capable de communiquer directement avec d'autres agents,
- capable de se mouvoir par un ensemble de tendances,
- possède des ressources propres,

- capable de percevoir (mais de manière limitée) son environnement,
- ne dispose que d'une représentation partielle de cet environnement (et éventuellement aucune),
- possède des compétences et offre des services,
- peut éventuellement se reproduire,
- son comportement tend à satisfaire ses objectifs, en tenant compte des ressources et des compétences dont elle dispose, et en fonction de sa perception, de ses représentations et des communications qu'elle reçoit.

À l'opposé, nombreux sont ceux qui considèrent un agent comme une "entité autorisée à agir au nom de quelqu'un d'autre". Une telle définition met sur le même plan un agent intelligent, un agent de police, un agent de sécurité ou un agent commercial. En conséquence, la distinction entre un agent intelligent et un simple logiciel demeure très floue.

Malgré ses limitations, cette vision constitue un point de départ pour une définition qui soit suffisamment réaliste sans être réductrice. On peut ainsi affirmer qu'un agent intelligent est une entité logicielle qui possède des attributs propres et qui agit dans le but d'accomplir un certain nombre de tâches au nom d'une autre entité (un autre agent ou une personne). Le problème est maintenant de définir les attributs propres à un agent et, sur ce point, les polémiques sont féroces.

Les principales caractéristiques d'un agent sont:

- *L'autonomie*: les agents opèrent sans intervention directe de l'être humain ou autre instance et ont néanmoins un certain contrôle sur leurs actions et leur état interne.
- *L'interactivité*: les agents interagissent avec les autres agents ainsi qu'avec les humains.
- *La réactivité*: les agents perçoivent leur environnement qui peut être soit le monde physique, soit un utilisateur via une interface graphique, soit l'Internet, soit toutes ces modalités à la fois. Ils répondent donc aux changements qui apparaissent.

- *Le comportement intentionnel*: les agents ne réagissent pas simplement en réponse à leur environnement, ils sont capables d'avoir un comportement dirigé vers un but et de prendre des initiatives.
- *La capacité d'apprentissage*: l'agent est capable de s'adapter aux directives de son utilisateur en analysant et interprétant ses actions passées.
- *La flexibilité*: les actions d'un agent ne sont pas entièrement préétablies. L'agent est en effet capable de choisir ce qu'il va entreprendre et selon quelle séquence, en fonction de l'environnement externe.
- *L'auto-démarrage*: contrairement aux logiciels traditionnels, un agent peut décider du moment précis pour entamer une action toujours en fonction de l'environnement externe.
- *La mobilité*: certains agents peuvent être stationnaires comme le modèle traditionnel client-serveur. Ils résident soit sur la machine de l'utilisateur, soit sur le serveur. D'autres agents peuvent être mobiles comme l'illustre la figure 3.1, c'est-à-dire qu'ils se baladent sur le réseau. Ils peuvent se déplacer d'une machine à une autre pendant l'exécution des instructions en amenant avec eux leur environnement d'exécution. Ces agents peuvent se présenter à d'autres agents qui peuvent leur fournir certains services, ou servir de point de rencontre entre différents agents. Nous avons vu dans la description des différents degrés de mobilité que la migration forte permet de déplacer un processus, ou un agent, avec son code (s'il n'est pas disponible sur la machine destinataire), ses données et son contexte d'exécution. Ainsi, après une migration, l'exécution se poursuit au point où elle s'est arrêtée à la machine source. Cependant, ce type de migration, qui se base sur une capture de l'état d'exécution d'un agent, est difficile et coûteuse à réaliser surtout dans les environnements hétérogènes où les agents vont évoluer. Il faut faire correspondre la capture d'un état d'exécution d'un agent dans les différents interprètes.

Pour certains chercheurs, selon les conceptions actuelles de l'intelligence artificielle (IA), le terme "agent" a un sens plus fort et plus spécifique. Pour eux, un agent est un système qui, en plus des propriétés précédentes, est conceptualisé selon des notions que l'on attribue plus couramment aux humains. Par exemple, il est courant en intelligence artificielle de caractériser

un agent par des notions purement cognitives comme les connaissances, les convictions, les intentions ou les obligations. Certains chercheurs sont allés plus loin et parlent d'agents émotionnels.

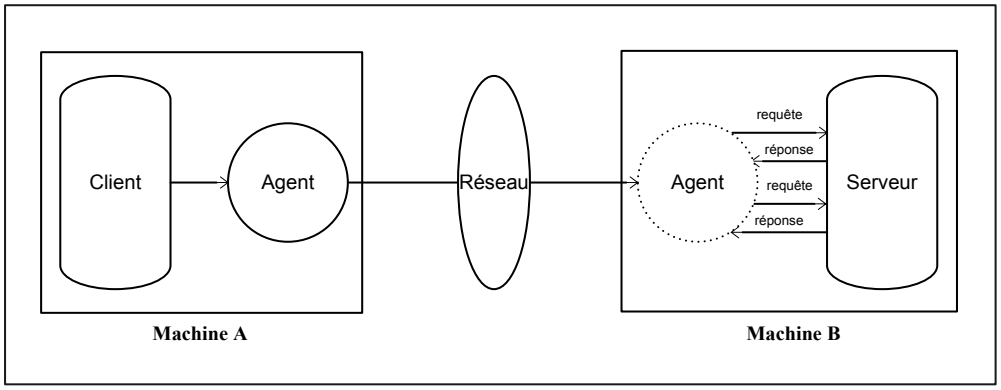


Figure 3.1 Modèle de l'agent mobile

À partir de quel moment peut-on parler d'un agent intelligent? Faut-il qu'il possède tout ou une partie de ces attributs? Le débat est interminable, peut-être insoluble et sûrement sans grand intérêt pour l'utilisateur final.

Dans le contexte du présent travail, un agent mobile est défini comme une entité informatique capable de raisonner, de naviguer sur l'infrastructure réseau pour s'exécuter sur un autre site éloigné, de chercher et de rassembler les résultats, de coopérer avec d'autres agents et de retourner au site d'origine après avoir complété la tâche assignée.

3.2 Efforts de normalisation des plateformes d'agents mobiles

Depuis l'apparition du concept d'agents mobiles, des nombreuses plateformes telles que Aglets [Lange et Oshima, 1998], JADE [Bellifemine at al., 2007] et Voyager [Recursion Software, 2011] ont été développées pour faciliter la mise en œuvre des applications à base de ce paradigme. L'architecture et la mise en œuvre de ces systèmes sont très différentes. Ceci empêche l'interopérabilité entre les plateformes et freine ainsi la diffusion rapide de la technologie d'agents mobiles. Pour promouvoir l'interopérabilité et la diversité du système, plusieurs organisations ont proposé des standards. Deux d'entre elles sont la norme de *Mobile*

Agent System Interoperability Facility (MASIF) [Milojicic et al., 1998] et celle de *Foundation for Physical Intelligent Agents (FIPA) [IEEE Foundation for Intelligent Physical Agents, 2011]*.

3.2.1 Norme MASIF

Dans cette section, nous allons présenter une synthèse de la norme *MASIF* qui a pour objectif de spécifier une architecture permettant l'interopérabilité entre les plateformes. La norme *MASIF* est définie par *Object Management Group (OMG)* qui est une organisation internationale appuyée par plus de 800 membres dont des fournisseurs de systèmes d'information, des développeurs et des utilisateurs de logiciels. Toutefois une redéfinition des concepts d'agents selon la norme *MASIF* s'impose au préalable.

Interopérabilité

L'interopérabilité devient réalisable si les actions comme le transfert d'agents, le transfert de classes et la gestion d'agents sont normalisés. Lorsque les systèmes d'agents source et destination sont similaires, la normalisation de ces actions peut aboutir à l'interopérabilité. Cependant, lorsque les deux systèmes d'agents sont radicalement différents, seulement une interopérabilité minimale peut être réalisée.

La question qu'on pourrait se poser est: qu'est-ce qui devrait être normalisé? La norme MASIF propose de standardiser les fonctionnalités suivantes [Object Management Group, 2000] [Milojicic et al., 1998]:

- La *gestion d'agents*: Il s'agit de prévoir un administrateur des systèmes d'agents pour les opérations suivantes: la création, la suspension, la reprise et la terminaison d'agents.
- Le *transfert d'agents*: Il est souhaitable que les agents se déplacent entre les différentes plateformes d'agents. Ceci résulte d'une infrastructure commune, et par conséquent un plus grand nombre des systèmes d'agents disponibles pour une éventuelle migration.
- L'*attribution d'un nom à l'agent et au système d'agents;*

- Le *type d'agents et la syntaxe associée au déplacement*: L'agent peut être transféré si le système d'accueil le supporte. La syntaxe d'emplacement est aussi standardisée pour que les plateformes et les agents s'identifient mutuellement.

Ces fonctionnalités sont concrètement spécifiées par la norme MASIF en définissant en IDL les deux interfaces suivantes:

1. *MAFAgentSystem* qui définit les opérations d'agents telles que la réception, la création, la suspension et la terminaison;
2. *MAFFinder* qui définit les opérations de gestions telles que l'enregistrement, la libération, la localisation des agents, les places et les systèmes d'agents.

La figure 3.2 nous présente les deux interfaces proposées par la norme. MASIF ne produit pas de code, elle ne fournit que des spécifications. L'implémentation (le passage de la spécification au programme) est laissée aux développeurs des systèmes. La norme MASIF, dans sa forme actuelle, offre les caractéristiques requises pour un premier niveau d'interopérabilité de la mobilité de l'agent. Une fois que l'agent est transféré d'un système à l'autre, la façon dont les systèmes le traitent avec ses paramètres internes est une question de mise en œuvre. Celle-ci est laissée aux développeurs des systèmes d'agents.

Agent

Actuellement, la plupart des agents sont programmés en langage interprété (par exemple en Java) pour des raisons de portabilité. L'agent peut être soit stationnaire soit mobile.

Agent stationnaire

Un agent stationnaire s'exécute seulement sur le système où il commence son exécution. Si l'agent a besoin des données qui ne sont pas sur ce système, il utilise généralement un mécanisme de communication tels que l'appel de procédure à distance (en anglais *Remote Procedure Calling, RPC).*

Agent mobile

Un agent mobile n'est pas figé au système où il commence son exécution. Il a la capacité unique de se déplacer d'un système à un autre dans le réseau. Cette spécification est principalement écrite pour les agents mobiles.

```
interface MAFFinder {
        void register_agent (
                in Name agent_name,
                in Location agent_location,
                in AgentProfile agent_profile) raises (NameInvalid);
        void register_agent_system (
                in Name agent_system_name,
                in Location agent_system_location,
                in AgentSystemInfo agent_system_info) raises (NameInvalid);
        void register_place (
                in string place_name,
                in Location place_location) raises (NameInvalid);
        Locations lookup_agent (
                in Name agent_name,
                in AgentProfile agent_profile) raises (EntryNotFound);
        Locations lookup_agent_system (
                in Name agent_system_name,
                in AgentSystemInfo agent_system_info)
                        raises (EntryNotFound);
        Locations lookup_place (in string place_name)
                        raises (EntryNotFound);
        void unregister_agent (in Name agent_name)
                        raises (EntryNotFound);
        void unregister_agent_system (in Name agent_system_name)
                        raises (EntryNotFound);
        void unregister_place (in string place_name)
                        raises (EntryNotFound);
};
```

(a) interface MAFFinder

```
interface MAFAgentSystem {
        Name create_agent (in Name agent_name,
        in AgentProfile agent_profile,
        in OctetString agent,
        in string place_name,
        in Arguments arguments,
        in ClassNameList class_names,
        in string code_base,
        in MAFAgentSystemclass_provider) raises (ClassUnknown,
                ArgumentInvalid, DeserializationFailed,
                MAFExtendedException);
        OctetStrings fetch_class(in ClassNameList class_name_list,
                in string code_base, in AgentProfile agent_profile) raises
                (ClassUnknown, MAFExtendedException);
        Location find_nearby_agent_system_of_profile (in AgentProfile
                profile) raises (EntryNotFound);
        AgentStatus get_agent_status(in Name agent_name)
                raises(AgentNotFound);
        AgentSystemInfo get_agent_system_info();
        AuthInfo get_authinfo(in Name agent_name) raises
                (AgentNotFound);
        MAFFinder get_MAFFinder() raises (FinderNotFound);
        NameList list_all_agents();
        NameList list_all_agents_of_authority(in Authority authority);
        Locations list_all_places();
        void receive_agent(in Name agent_name,
                in AgentProfile agent_profile,
                in OctetString agent,
                in string place_name,
                in ClassNameList class_names,
                in string code_base,
                in MAFAgentSystemagent_sender) raises (ClassUnknown,
                ArgumentInvalid, DeserializationFailed,
                MAFExtendedException);
        void resume_agent(in Name agent_name) raises (AgentNotFound,
                ResumeFailed, AgentIsRunning);
        void suspend_agent(in Name agent_name) raises (AgentNotFound,
                SuspendFailed, AgentIsSuspended);
        void terminate_agent(in Name agent_name) raises (AgentNotFound,
                TerminateFailed);
        };
};
```

(b) interface MAFAgentSystem

Figure 3.2 Interface de la norme MASIF [Object Management Group, 2000]

État d'un agent

Lorsqu' un agent se déplace d'un *système source* à un *système destination*, son état et son code sont transportés avec lui. Grâce à son état qui migre avec lui, l'agent reprend son exécution là où il a été interrompu sur le *système source*.

Autorité de l'agent

L'autorité d'un agent identifie la personne ou l'organisation pour qui l'agent agit. Une autorité doit être authentifiée.

Nom d'agent

Un agent nécessite un nom qui l'identifie lors des opérations de gestion et il peut être localisé via un service de nommage. Les noms des agents sont définis par une combinaison de leur autorité, de leur identité et leur type de système d'agents. La combinaison de l'autorité, de l'identité de l'agent et le type de système d'agents est toujours une valeur globalement unique. Comme le nom d'un agent est globalement unique et immuable, il peut être utilisé comme une clé dans les opérations qui se rapportent à une instance d'un agent particulier.

Emplacement d'agents

L'emplacement d'un agent est l'adresse d'une place. Cette dernière réside dans un système d'agents. Par conséquent, un emplacement d'un agent devrait contenir le nom du système d'agents où il réside et le nom de la place. Notons que si l'emplacement ne contient pas le nom de la place, le système d'agents de destination choisit une place par défaut.

Système d'agents

Un système d'agents est la plateforme qui fournit les mécanismes de création, d'interprétation, d'exécution, de transfert et de terminaison de l'agent. Comme un agent, un système d'agents est associé à une autorité qui identifie la personne ou l'organisation pour qui le système d'agents devrait agir. Par exemple, un système d'agents avec l'autorité de Bob peut mettre en œuvre la politique de sécurité de Bob pour la protection des ressources de Bob. Un système d'agents est identifié d'une manière unique par son nom et son adresse. Un hôte peut contenir un ou plusieurs systèmes d'agents. Chaque système d'agents possède un type (par exemple, *Voyager*) et ne peut que créer et supporter les agents de ce type. Les actions communes des systèmes d'agents sont:

- La création d'agents;

- L’attribution d’un nom et d’une adresse uniques aux agents;
- Le transfert et la réception d’agents;
- Le support du concept de région;
- La localisation d’agents;
- La sécurisation d’environnements d’exécution d’agents.

Lors de la migration d’un agent, celui-ci crée une requête de transfert qui contient son nom et son adresse de destination. L'agent spécifie aussi la qualité de service de communication requise pour son transfert. Ce service n'est pas spécifié par la norme *MASIF*, il est laissé aux développeurs des systèmes d’agents. Une fois que le système d’agents de destination donne son accord, l'état de l'agent, l'autorité, les lettres de créance de sécurité et, si nécessaire, son code sont transférés à la destination. Le système d’agents de destination réactive l'agent et son exécution continue son cours. Lors du transfert de l’agent, le système source effectue les étapes suivantes:

1. L’interruption de l'agent;
2. La capture de l'état de l'agent à transférer;
3. La sérialisation de l’instance de la classe agent et son état;
4. Le codage de l’agent sérialisé en choisissant le protocole de transport;
5. L’authentification du système de destination;
6. Le transfert de l'agent.

La figure 3.3 présente un système d’agents. Avant qu'un agent ne soit reçu dans un système de destination, celui-ci doit déterminer s'il peut l’interpréter. Si c’est le cas, le système de destination accepte l'agent et procède aux actions suivantes:

1. L’authentification du système source;
2. Le décodage de l'agent;
3. La désérialisation de la classe agent et son état;
4. L’instanciation de l'agent;
5. La restauration de l'état d'agent;
6. La reprise de l'exécution d'agent.

Sérialisation/Désérialisation

La sérialisation est un processus qui permet d'encoder l'état d'un agent sous la forme d'une suite d'octets. Cette suite est utilisée pour la sauvegarde (persistance), ensuite l'envoie dans le réseau. La désérialisation est le processus symétrique permettant de décoder cette suite pour créer une copie conforme de l'état de l'agent dans sa forme originale. Notons que cette suite d'octets doit être capable d'identifier et de vérifier les classes dont les champs ont été sauvegardés.

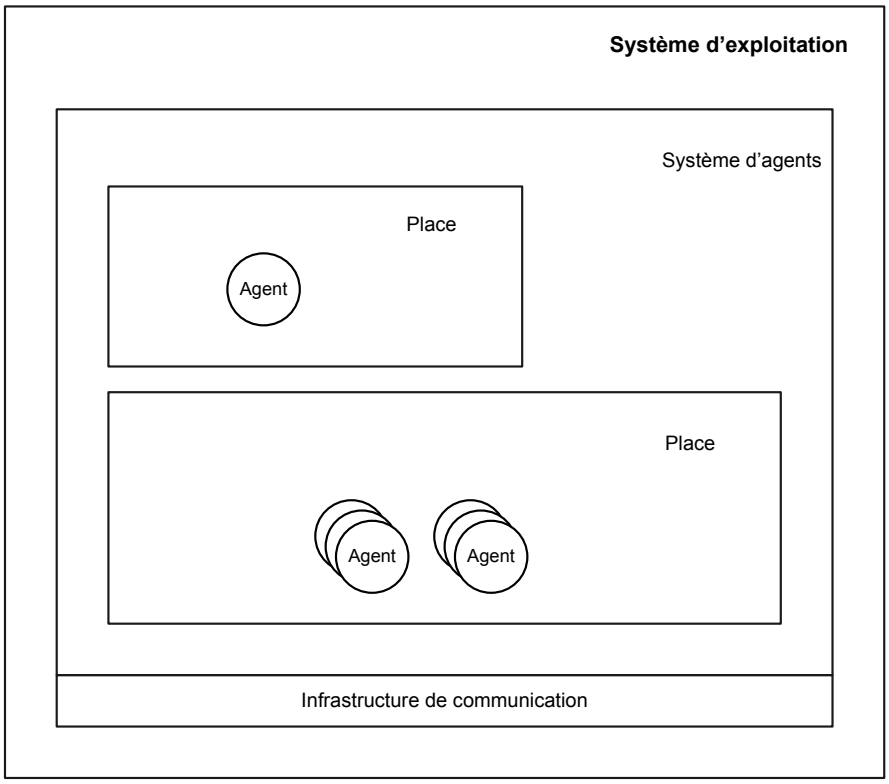


Figure 3.3 Système d'agents [Object Management Group, 2000]

Communication entre les systèmes d'agents

La communication entre les systèmes d'agents passe via une infrastructure de communication (IC). Celle-ci fournit les services de transport de communications (par exemple, RPC), le nommage et les services de sécurité pour un système d'agents. L'administrateur de région définit les services de communication inter et intra régionale. La figure 3.4 présente la communication entre les systèmes d'agents.

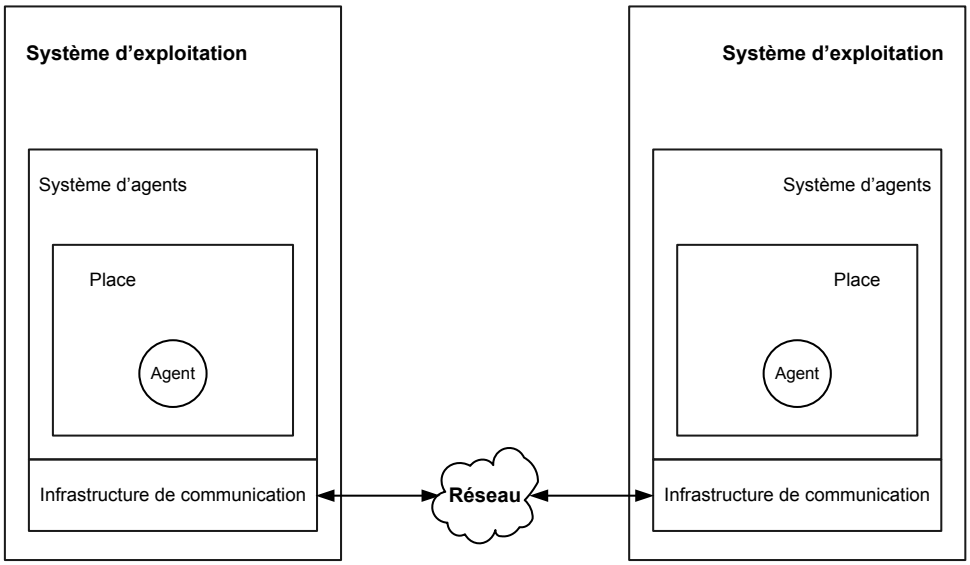


Figure 3.4 Communication entre les systèmes d'agents [Object Management Group, 2000]

Place

Lors de la migration, un agent se déplace entre des environnements d'exécution appelés places. Une place est un contexte dans un système d'agents dans lequel un agent peut s'exécuter. Ce contexte peut fournir des fonctions comme le contrôle d'accès. Les places source et destination peuvent être dans un même système ou dans différents systèmes qui supportent le même profil d'agents.

Une place est associée à un emplacement qui se compose du nom de la place et de l'adresse du système d'agents. Un système d'agents peut contenir une ou plusieurs places. Une place peut contenir un ou plusieurs agents. Lorsqu'un client demande l'emplacement d'un agent, il reçoit l'adresse de la place où l'agent s'exécute.

Région

Une région peut contenir un ou plusieurs systèmes d'agents qui ont la même autorité mais ne sont pas nécessairement du même type de système d'agents. Le concept de région permet à une personne ou à une organisation d'être représentée par un ou plusieurs systèmes d'agents. Les régions permettent l'adaptabilité en distribuant la charge à travers des multiples systèmes d'agents.

Une région fournit un niveau d'abstraction aux clients communiquant avec d'autres régions. Il est possible de communiquer avec des agents distants exactement de la même manière que s'ils se trouvaient au sein de la même région. Un client peut communiquer avec un agent distant en sachant seulement son nom et son adresse peu importe son emplacement.

Un agent peut aussi avoir la même autorité que la région dans laquelle il réside. Cela signifie que l'agent représente la même personne ou la même organisation que la région. Cette configuration de la région peut ainsi accorder plus de privilèges à un tel agent qu'à un autre avec une autorité différente. Par exemple, on peut accorder à un agent qui a la même autorité que la région des privilèges administratifs. Dans une région, les systèmes d'agents sont entièrement connectés entre eux. Ceci permet le transfert d'information point à point. Chaque région contient un ou plusieurs points d'accès de région et par ces biais, les régions sont donc connectées pour former un réseau. La figure 3.5 présente l'architecture d'une région.

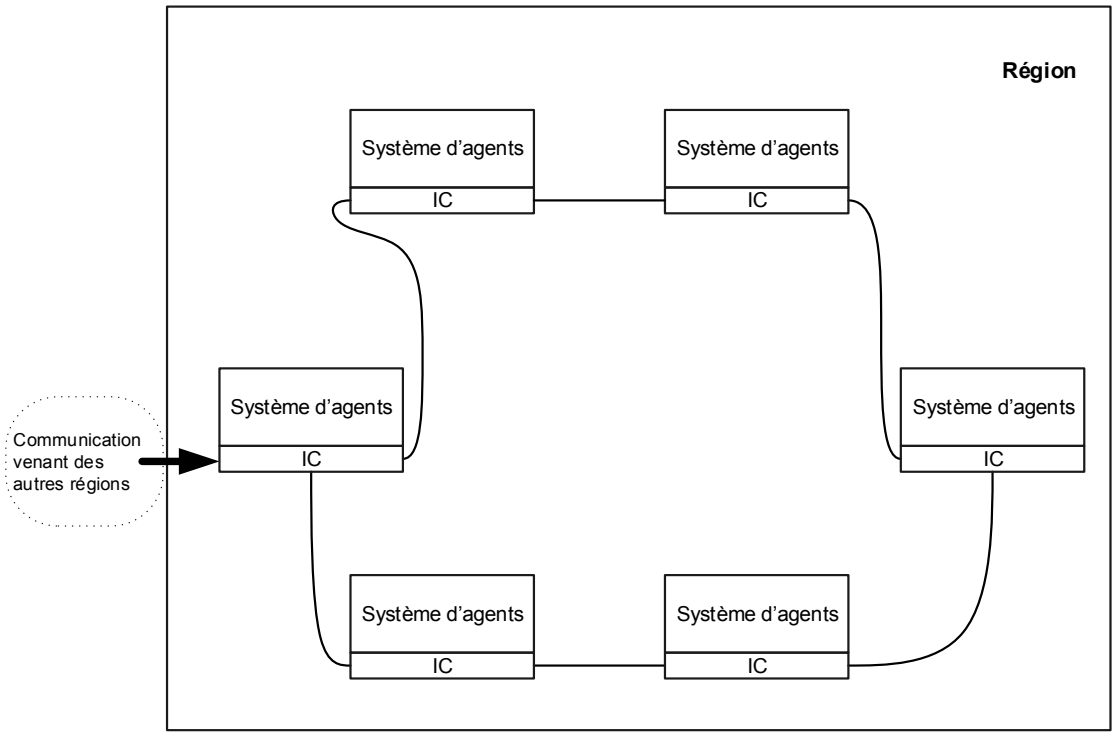


Figure 3.5 Architecture d'une région [Object Management Group, 2000]

Communication entre les régions

Les régions communiquent via un ou plusieurs réseaux et peuvent partager un même service qui leur attribue un nom basé sur un accord entre des autorités de région. Un système ne supportant pas d'agents peut aussi communiquer avec les systèmes d'agents dans n'importe quelle région pour autant que ce système ait l'autorisation de faire ainsi. Une région contient un ou plusieurs systèmes d'agents. Ces derniers et les clients externes ont accès à la région via les systèmes d'agents qui sont exposés au monde extérieur. Cette configuration est semblable à une situation de coupe-feu. Ces systèmes d'agents sont définis comme des points d'accès de région.

3.2.2 Normes FIPA

FIPA est une organisation internationale dont l'objectif est de produire des spécifications pour faciliter l'interopérabilité entre les agents logiciels développés par différents constructeurs dans un «système ouvert». Ces spécifications ne décrivent pas l'architecture interne de l'agent ou sa mise en œuvre mais elles définissent plutôt les interfaces nécessaires pour supporter l'interopérabilité entre les différents systèmes d'agents.

Architecture abstraite

L'architecture abstraite de la plateforme supportant le protocole de communication *FIPA* est définie par le modèle de référence de gestion d'agents. Ce modèle fournit un cadre normatif dans lequel les agents *FIPA* évoluent. Il établit un contexte pour la création, l'enregistrement, la localisation, la communication, la migration et la suppression des agents. Le modèle de référence de gestion d'agents est composé des éléments suivants:

- L'*agent* qui demeure un processus autonome capable de communiquer avec son environnement. Les agents utilisent un langage de communication d'agents *(Agent Communication Language* ou *ACL)*. L'agent est l'acteur fondamental sur la plateforme. Il combine une ou plusieurs capacités de service à l'intérieur d'un modèle d'exécution unifié et intégré. Dans le modèle de référence, l'agent est identifié par son nom de façon unique dans sa communauté.

- Le *service de répertoire (Directory Facilitator* ou *DF):* un composant optionnel de la plateforme qui fournit des services de «pages jaunes» aux agents. Ces derniers peuvent enregistrer leurs services auprès du *DF* ou interroger le *DF* pour découvrir les services offerts par les autres agents.
- Le *système de gestion d'agents (Agent Management System* ou *AMS)*: un composant obligatoire de la plateforme qui contrôle l'accès et l'utilisation de la plateforme. Un seul *AMS* réside dans chaque plateforme. *AMS* maintient une liste d'adresses de tous les agents enregistrés.
- Le *service de transport de messages (Message Transport Service* ou *MTS)* est la méthode de communication par défaut entre les agents des différentes plateformes.
- La *plateforme d'agents* fournit l'infrastructure physique dans laquelle les agents peuvent être déployés. Une plateforme est composée d'une machine, d'un système d'exploitation, d'un logiciel du support d'agents, des composants de gestion d'agents *FIPA (DF, AMS et MTS)* et des agents. *FIPA* spécifie la communication entre les agents natifs des plateformes pour utiliser toute méthode propriétaire d'intercommunication.
- Le *logiciel* décrit tous ceux qui ne sont pas des agents. Il s'agit des collections d'instructions exécutables accessibles par un agent. Les agents peuvent accéder à des logiciels afin par exemple:
    - d'ajouter de nouveaux services,
    - d'acquérir de nouveaux protocoles de communications,
    - d'acquérir de nouveaux protocoles/algorithmes de sécurité,
    - d'acquérir de nouveaux protocoles de négociation,
    - d'avoir accès aux outils qui supportent la migration, etc.

Contrairement à un document qui laisse entendre que la migration n'est plus supportée, après vérification, nous pouvons confirmer que les normes FIPA supportent bel et bien la mobilité des agents [FIPA Agent Management Support for Mobility Specification, 2000]. En effet, il existe même un groupe de travail [Mobile Agents Working Group, 2005] dont l'objectif principal consiste à reprendre les travaux sur les normes d'agents mobiles. Les spécifications existantes seront ainsi améliorées/étendues, les résultats de recherche et les implémentations

existantes y seront intégrés. Avec des propositions de spécification, le groupe de travail développera les mises en œuvre de protocoles de référence en forme de composants logiciels. La figure 3.6 présente le modèle de référence de gestion d'agents.

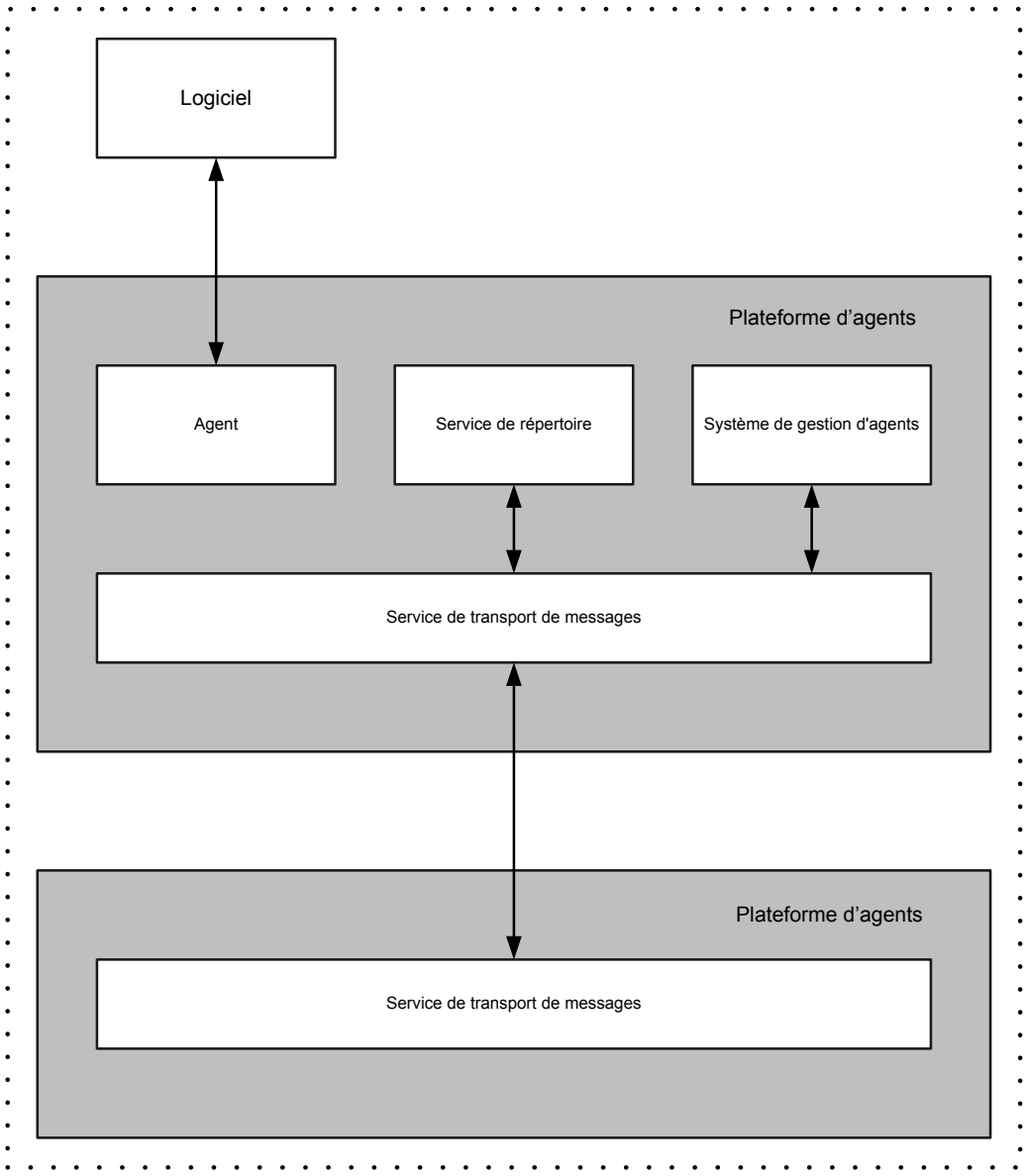


Figure 3.6 Modèle de référence de gestion d'agents

3.2.3 Synthèse de normes MASIF et FIPA

Les normes *MASIF* et *FIPA* ne sont qu'un point de départ pour permettre l'interopérabilité des plateformes d'agents et ainsi exploiter les paradigmes basées sur les agents mobiles. Nous constatons certaines similitudes et différences entre les normes FIPA et MASIF. En effet, les

efforts de normalisation MASIF et FIPA ne sont pas complètement différents. Les deux normes mettent l'accent sur l'interopérabilité entre les systèmes d'agents. Cependant, chacune a sa propre façon de réaliser cette interopérabilité. La norme MASIF utilise la mobilité d'agents tandis que FIPA se concentre sur la communication d'agents. En ce qui concerne les caractéristiques de deux normes, nous avons constaté que les deux sont complémentaires [Islam et al., 2010].

Les normes *MASIF* et *FIPA* ont apporté un gain considérable à la communauté des agents mobiles. Cependant, ces normes présentent des limites. En effet, aucune de ces deux normes ne spécifie la structure interne de l'agent mobile, par exemple. Bref, les deux normes présentent une spécification incomplète d'agents mobiles. Dans la prochaine section, nous allons présenter les formalismes d'agents mobiles qui tentent d'apporter des solutions à cette situation.

3.3 Formalismes d'agents mobiles

Comme les agents mobiles constituent un concept utilisé dans différents domaines, il est donc difficile d'en trouver une formalisation unique. Dans la littérature, il existe de nombreux formalismes pour la modélisation des systèmes à base d'agents mobiles. En effet, ces systèmes peuvent être modélisés à différents niveaux d'abstraction (structurelle versus comportementale) en utilisant des notations très variées (semi-formelle versus formelle).

3.3.1 Modélisation structurelle

Une modélisation structurelle permet d'illustrer l'architecture générale d'un système, ses composants ainsi que ses différents types de liens d'interactions. Dans [Muscutariu et Gervais, 2001], les auteurs ont, d'une part, défini un *ADL* (*Architecture Description Language*) permettant de modéliser l'infrastructure afin de supporter l'exécution des agents mobiles conformément au standard *MASIF* [Object Management Group, 2000]. Ils ont, d'autre part, spécifié les éléments nécessaires pour la représentation des mécanismes permettant une transparence à la distribution comme l'accès, la persistance, la migration des

agents et leur localisation. *ADL* est défini comme étant un profil *UML (Unified Modeling Language)* nommé «*MASIF-DESIGN*» qui spécifie les concepts de «région», «cœur d'agence» par des sous-systèmes stéréotypés.

Une plateforme conforme à la norme *MASIF* définit les éléments suivants: une région, un système d'agents, un cœur d'agence, une place et un agent. Chaque élément de la plateforme offre un ensemble des opérations qui représente les interactions entre les éléments. Ces interactions forment le raffinement nécessaire pour une spécification comportementale, le raffinement qui détaille la spécification opérationnelle d'un système d'agents.

La région est un système d'enregistrement supportant la localisation. Elle est modélisée comme un sous-système stéréotypé. En plus de l'interface *MAFFinder* spécifié dans *MASIF*, deux autres groupes d'opérations peuvent être fournis afin de faciliter les services de système d'agents. Comme la plateforme *Grasshopper* [Breugst et al., 1998], par exemple, qui fournit l'opération *lookupCommunicationServer()* permettant de connaître le mécanisme de communication sous-jacente que les agents utilisent (par exemple, socket, CORBA or RMI). Ces opérations sont mises en œuvre dans des méthodes avec le prototype défini dans l'interface *IRegion* (voir la figure 3.7).

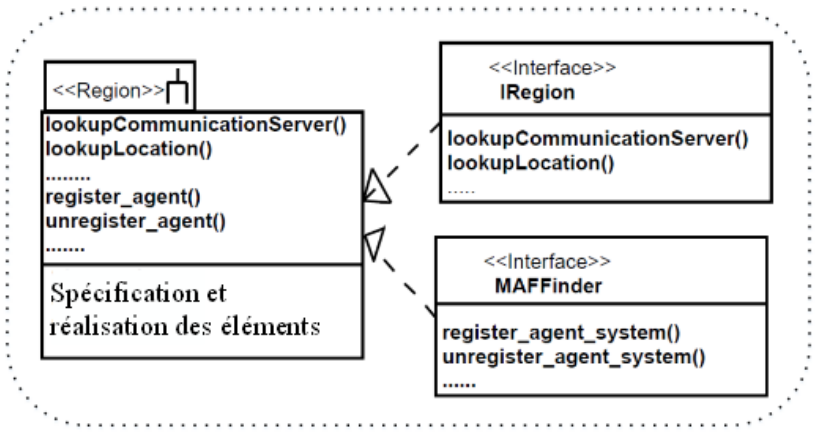


Figure 3.7 Sous-système d'une région stéréotypé [Muscutariu et Gervais, 2001]

Le système d'agents est la plateforme qui permet de créer, d'interpréter, d'exécuter, de transférer et de terminer les agents. Il est représenté comme un nœud stéréotypé. Chaque système d'agents a un cœur d'agence et un ou plusieurs places. Le système d'agents agit

comme un conteneur pour exécuter les agents en utilisant les fonctionnalités offertes par le cœur d'agence.

Le cœur d'une agence permet de mettre en œuvre la gestion d'agents. Il est modélisé par un sous-système stéréotypé. En plus des actions spécifiées dans l'interface *MAFAgentSystem* définie dans la norme MASIF, d'autres opérations peuvent être identifiées afin de permettre la surveillance et le contrôle des agents en cours d'exécution. Dans [Muscutariu et Gervais, 2001], une interface appélée IAgentSystem est ajoutée comme défini dans la plateforme Grasshopper [Breugst et al., 1998] (voir la figure 3.8). Un exemple d'opérations que l'on retrouve dans cette interface est la fonction *saveAgent()* qui enregistre les données de l'agent afin de permettre une éventuelle reprise.

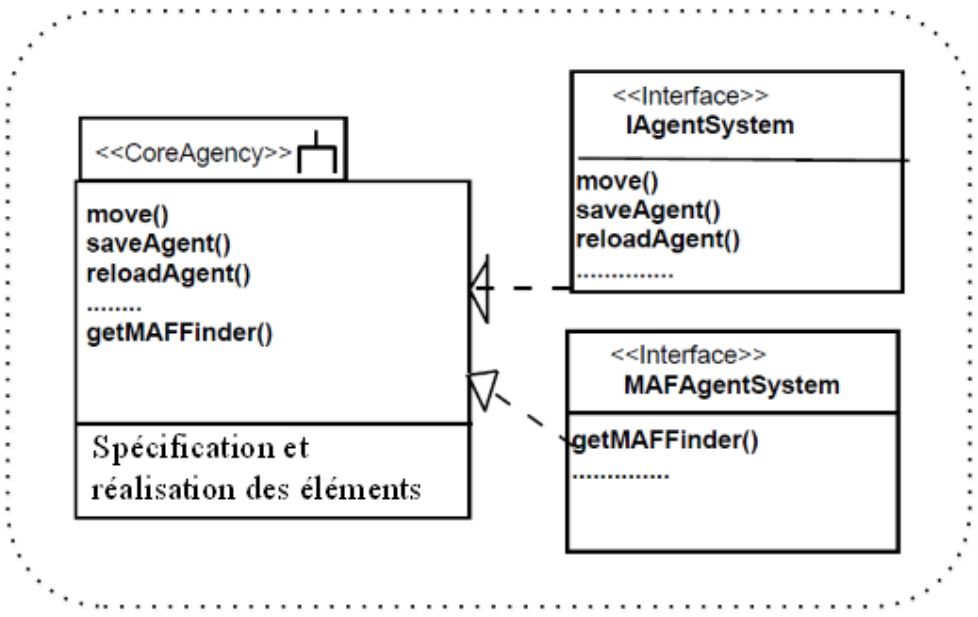


Figure 3.8 Sous-système d'un cœur d'agence stéréotypé [Muscutariu et Gervais, 2001]

Une *place* est un *contexte* qui est défini au sein d'un *système d'agents* dans lequel s'exécute un agent. Elle est modélisée par un *paquetage* et peut fournir des fonctions telles que le contrôle d'accès. Le concepteur peut définir, dans une interface nommée *SpecialPlaceInterface*, les opérations qui offrent des services aux agents situés dans une place *SpecialPlace*. Il ne peut y avoir qu'une seule *SpecialPlace* dans un système d'agents (voir la figure 3.9).

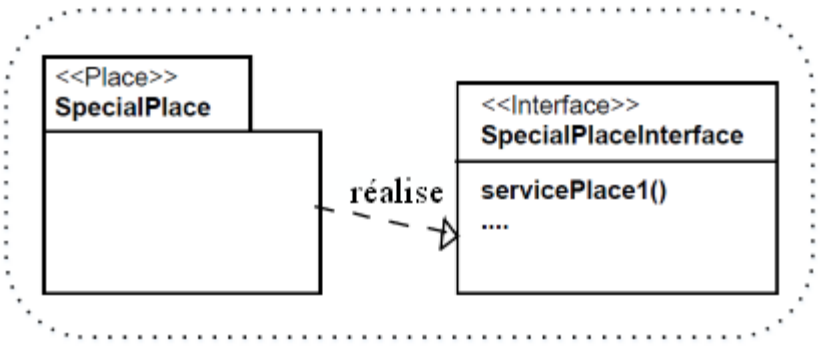


Figure 3.9 Paquetage SpecialPlace [Muscutariu et Gervais, 2001]

Un agent est représenté comme un élément stéréotypé. Des informations supplémentaires nécessaires pour la mise en œuvre peuvent être incluses dans le diagramme composant lié à une mise en œuvre d'agent. Encore une fois, l'exemple de la plateforme *Grasshopper* est utilisé pour identifier ces informations supplémentaires. Puisque la plateforme *Grasshopper* est un environnement pour un agent typé, la mise en œuvre d'agent modélisé comme un composant doit hériter d'une structure spécifique [Breugst et al., 1998]. Les opérations *beforeMove()* et *afterMove()* permettent de préparer la migration de l'agent (c'est-à-dire, la sauvegarde de l'état d'exécution). La figure 3.10 nous présente le diagramme de composant d'agent.

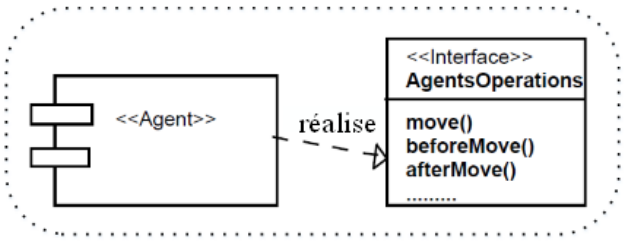


Figure 3.10 Diagramme de composant d'agent [Muscutariu et Gervais, 2001]

Dans [Chhetri et al., 2006], les auteurs proposent une spécification d'une ontologie pour la modélisation des systèmes à base d'agents mobiles. Dans leur article, les auteurs identifient des concepts clés pour la modélisation de systèmes d'agents mobiles. L'identification des principaux concepts et leurs interrelations est généralement une étape préalable à tout exercice de modélisation conceptuelle et reste le précurseur de la spécification formelle. Les concepts identifiés comme base pour décrire la mobilité d'agents sont: les *rôles,* les *agents,* le *domicile,* les *itinéraires,* les *visites,* les *tâches,* l'*emplacement,* les *contraintes de migration* et les

ressources. La figure 3.11 illustre les concepts principaux de la mobilité d'agents et leurs corrélations [Chhetri et al., 2006].

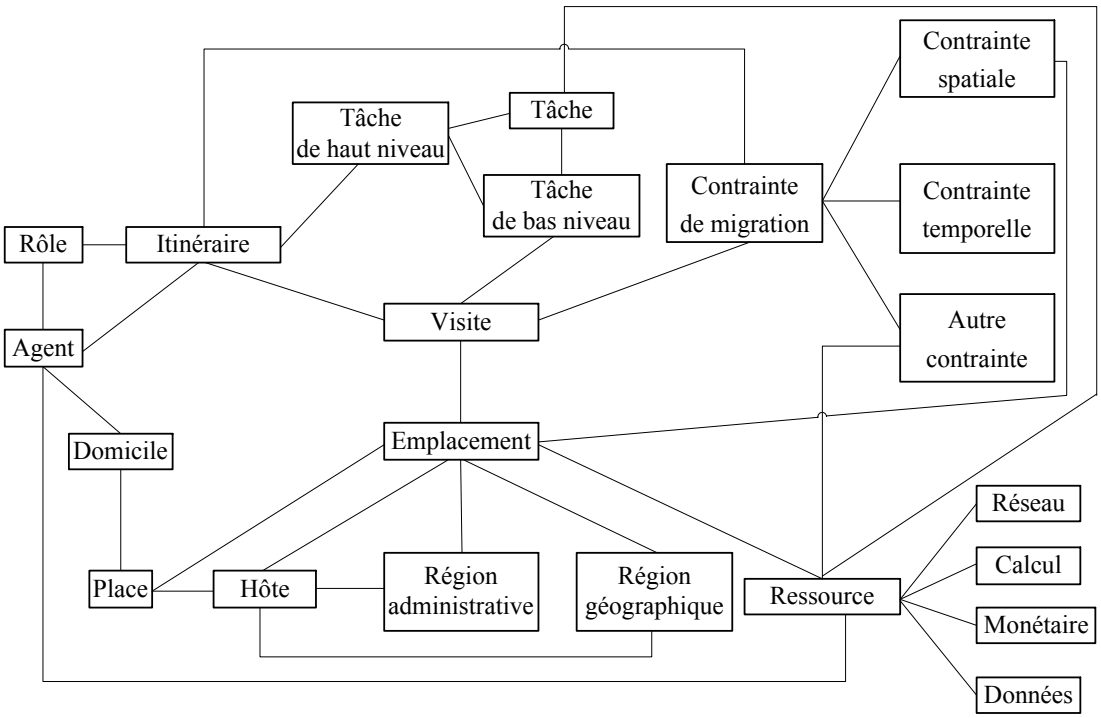


Figure 3.11 Concepts principaux de la mobilité d'agents et leurs corrélations

Rôles

Un rôle est par définition une position ou un but que n'importe quel objet/entité/agent a dans un système, une situation, une organisation, une société ou une relation. Typiquement, un agent dans un système doit acquérir ou prendre au moins un rôle et peut accomplir plusieurs rôles. Dans le contexte de la modélisation des systèmes d'agents mobiles, le rôle est perçu comme un concept principal parce qu'il peut exiger l'attribut de mobilité pour accomplir des responsabilités, des engagements et des obligations. Ainsi, un agent qui a un rôle et possède l'attribut de la mobilité sera un agent mobile. Le concept de rôle qui nécessite la mobilité doit être traité à la phase d'analyse pour développer des systèmes d'agents mobiles.

Agents

En général, un agent peut être défini comme une entité logicielle proactive, réactive, autonome et communicative placée dans un environnement. Dans le contexte d'agents utilisés pour des

applications mobiles et distribuées, la mobilité d'agents est évidemment un concept principal qui doit être modélisé. Comme dans le concept de rôle ou de tâches, la mobilité peut être associée à un agent. Cependant, même dans des cas où ni les rôles ni des tâches n'exigent d'attribut de mobilité, une spécification logicielle peut exiger que les agents soient mobiles pour faire face aux changements inattendus de la disponibilité de ressource (la migration obligatoire dans le cas d'échec hôte, par exemple).

Domicile

La notion de domicile est explicitement prise en compte dans plusieurs systèmes d'agents mobiles. Le *domicile* fait référence à l'origine ou au lieu de la création d'un agent mobile. Tout agent mobile ou fixe a bel et bien un «foyer» mais cette notion est particulièrement importante lorsqu'un agent possède la capacité de se déplacer et de visiter plusieurs endroits. Dans ce modèle, le *domicile* est un sous-type de lieu modélisé séparément car il possède des attributs supplémentaires tels que la transmission de messages aux agents et peut avoir des spécifications de contraintes distinctes. Les agents, qui ont des contraintes spatiales comme domicile, doivent être dans une certaine région administrative, par exemple.

Itinéraires

Un itinéraire définit généralement un ordre de visites à divers endroits pour accomplir des tâches spécifiques (à savoir un bas niveau ou sous-tâche). Un itinéraire peut aussi être directement associé à une tâche ou un objectif (c'est à dire un travail de haut niveau nécessitant l'agent de migrer vers certains endroits d'une façon commandée), qui peut alors être précisée en termes de sous-tâches spécifiques à effectuer lors des visites individuelles. En outre, un itinéraire est associé aux contraintes de migration qui déterminent l'ordre de visites dans un itinéraire, la raison et le moment où un itinéraire particulier est choisi et entrepris. Pour assurer que la conception de mobilité est à un niveau abstrait et supporte ainsi la gamme complète d'applications d'agents mobiles, il est essentiel que le concept d'itinéraires soit traité comme un concept de modélisation orthogonal. Cela signifie que les itinéraires et, par conséquent, la mobilité peuvent être associés librement à un autre composant des systèmes d'agents qui pourraient exiger l'attribut de mobilité des agents, des rôles et des tâches.

Visites

Comme discuté dans le cadre des itinéraires, une visite se réfère à un arrêt dans un itinéraire. Une visite implique un emplacement spécifique d'une ou de plusieurs sous-tâches qui doivent être exécutées à cet emplacement. En outre, la migration d'un emplacement vers un autre peut être dictée par des contraintes de migration de niveau de visite (c'est-à-dire spécifiée au niveau d'une visite individuelle) mais aussi par des contraintes de migration indiquées pour l'itinéraire dans l'ensemble.

Tâches

Les tâches sont des activités que l'agent est tenu d'accomplir. Elles sont modélisées de haut niveau ou de bas niveau (des sous-tâches). Une tâche de haut niveau exige normalement la migration et décrit un but ou un travail. Une sous-tâche n'exige pas normalement de mobilité et décrit une activité à être exécuté dans une visite.

Emplacement

Le concept d'emplacement fait partie intégrante de systèmes d'agents mobiles. La plupart des méthodes de modélisation d'agents mobiles tels que *mGAIA (Mobile Generic Architecture for Information Availability)* et *AUML (Agent UML)* [Poggi et al., 2003] comprennent des notations pour modéliser l'emplacement. *mGAIA*, une extension de la méthodologie *GAIA*, ne traite pas la mobilité dans la phase d'analyse mais elle modélise la mobilité dans la phase de conception [Sutandiyo et al., 2004]. Comme avec Agent UML, qui ne respecte pas le concept orthogonal, le principal inconvénient de cette approche réside dans le fait que seule la mobilité en association avec un agent spécifique est prise en compte. Une nouvelle similitude avec Agent UML est que mGAIA ne développe pas entièrement le concept de visite [Chhetri et al., 2006].

mGAIA et *AUML* limitent la portée de la place (connu comme l'agence ou le contexte) en modélisant l'emplacement. Une place est l'environnement d'exécution qui permet d'héberger les agents mobiles et supporte leurs interactions avec des systèmes informatiques externes. Les auteurs prennent, dans le cas présent, une vue holistique d'emplacement où il est essentiel

de modéliser des domaines géographiques/administratifs et des hôtes (aussi connu comme des nœuds ou des dispositifs) en plus des places. La modélisation de l'emplacement en ces termes permet de faciliter un plus haut degré de flexibilité et de puissance expressive lors de spécification de l'emplacement, que ce soit dans le cadre d'une visite, une ressource ou une contrainte de migration spatiale. Les concepteurs peuvent ainsi faire varier la précision (en spécifiant la place, l'hôte et la région géographique) et la complexité des spécifications d'accès (en spécifiant l'hôte mobile qui est localisé pendant les périodes dans une région d'administration et/ou géographique spécifique).

Notons que ce modèle de localisation représente explicitement la possibilité de se déplacer entre des lieux d'un hôte donné ainsi que la migration généralement considéré entre les hôtes. Il permet en outre la possibilité d'hôtes et leurs lieux associés ayant la mobilité, par exemple, les appareils mobiles tels que les assistants numériques personnels (PDA) ou des lieux d'exécution sur PDA. Ceci est important car, de plus en plus, les applications d'agents mobiles sont considérées comme étant particulièrement adaptées pour les environnements «*pervasifs*» qui impliquent des dispositifs mobiles ayant des ressources limitées. Les droits/permissions d'accès seraient un exemple de contraintes de migration spatiales pour une visite ou un itinéraire qui illustre bien l'utilisation de l'emplacement dans la spécification d'application d'agents mobiles. Un autre exemple serait la spécification indirecte d'une variété de contraintes spatiales pour un agent donné, un rôle, une tâche de haut niveau via l'utilisation d'un itinéraire associé à une contrainte spatiale accompagnante. En intégrant les notions complémentaires de régions géographiques/administratives et la place dans la modélisation d'emplacement, les auteurs proposent un paradigme de modélisation plus générale pour les agents mobiles.

Contraintes de migration

En spécifiant la mobilité par des visites et des itinéraires, les contraintes qui déterminent la migration et leurs corrélations avec d'autres concepts d'agents sont souvent ignorées. On note que ces contraintes peuvent être associées à une visite individuelle ainsi que des itinéraires dans son ensemble. Les contraintes de migration spécifient les priorités qui déterminent le

moment, la *destination* et la *raison* de la migration. Il existe trois types de contraintes (séparées ou combinées) qui peuvent conduire à la migration:

1. Spatial: Les contraintes spécifient la destination de la migration de l'agent;
2. Temporel: Les contraintes spécifient le moment de la migration de l'agent (y compris les limites de temps);
3. Autres: Ces contraintes comprennent toutes les autres conditions qui sont associés à la migration et peuvent inclure:
    - La population d'agents: Si le nombre d'agents à un endroit particulier dépasse un nombre déterminé, l'agent devrait migrer vers un autre endroit où il ferait face à moins de concurrence pour les ressources;
    - La indisponibilité des ressources: Si une ressource particulière n'est pas disponible ou accessible, l'agent devrait migrer;
    - L'exigence de tâche: Une tâche peut exiger que l'agent soit dans un endroit particulier. Ceci exige la migration de l'agent vers cet endroit;
    - La migration forcée: Un agent est forcé de quitter un endroit précis de façon inattendue, par exemple, dans le cas de défaillance de l'hôte.

Ressources

Ce concept est associé aux contraintes de migration, des tâches, des agents et des emplacements. Une tâche peut exiger des ressources spécifiques (par exemple la disponibilité d'une connexion de réseau pour réserver un film) et une contrainte de migration peut dépendre de la disponibilité de certaines ressources. Un agent ou un emplacement peut être associé à (c'est à dire avoir) des ressources spécifiques. Une ressource comprend:

1. Les ressources de calcul;
2. Les ressources de réseau;
3. Les ressources de données;
4. Les ressources monétaires (un agent doit avoir ce qui est comparable aux unités monétaires afin de payer les ressources informatiques qu'ils utilisent).

3.3.2 Modélisation comportementale

La modélisation comportementale met l'accent sur la manière dont les agents mobiles évoluent et interagissent dans leur environnement réparti. Dans [Kusek et Jezic, 2005], les auteurs utilisent des diagrammes de séquence d'UML pour modéliser la mobilité d'agents en s'appuyant sur trois concepts clés d'agents: l'emplacement courant, le trajet de mobilité et l'emplacement de création.

Dans [Loukil et al., 2006], les auteurs proposent une notation MA-UML (Mobile Agent UML) pour modéliser l'aspect dynamique des agents mobiles. MA-UML permet des extensions des diagrammes d'activités, d'état-transition et de séquence d'AUML (Agent UML). Pour établir le lien entre un agent et son emplacement, le concept de localisation a été introduit au niveau du diagramme d'activités. MA-UML étend le diagramme d'état-transition pour supporter la modélisation d'un plan d'itinéraire des agents. En plus, MA-UML étend le diagramme de séquence d'AUML afin de supporter la spécification des différentes interactions qui peuvent se présenter entre les agents mobiles, les agents stationnaires, les places (emplacement d'exécution des agents mobiles) et les ressources.

D'autres travaux s'intéressent à des notations mathématiques (des méthodes formelles) pour formaliser l'aspect comportemental des agents mobiles. Les méthodes formelles sont des techniques permettant de raisonner rigoureusement, à l'aide de logique mathématique, sur les systèmes, afin de démontrer leur validité par rapport à une certaine spécification. Dans ce contexte, la formalisation d'agents mobiles fait appel d'une part aux systèmes de transitions [McCann et Roman, 1998] tels que les réseaux de Petri de haut niveau et d'autre part aux algèbres de processus avec comme base le π-calcul. Dans [Sewell et al., 1999], une communication entre agents mobiles est formalisée en utilisant une extension du π-calcul. Du point de vue de la mobilité, on peut aussi raisonner sur la localisation des agents. Ainsi dans le langage Klaim [Bettini et al., 2002], les propriétés des agents sont exprimées dynamiquement en fonction de l'évolution de leur localisation. Dans sa thèse, Loulou [Loulou, 2010] a proposé une spécification formelle des systèmes d'agents mobiles basée sur la notation Z [Woodcock et Davies, 1996].

3.4 Plateformes existantes

La très grande majorité des plateformes d'agents mobiles actuelles utilise Java grâce à sa machine virtuelle largement répandue qui cache l'hétérogénéité des processeurs et des systèmes d'exploitation sous-jacents. Java fournit un mécanisme pour la sérialisation [Grosso, 2001] et le chargement dynamique des classes qui sont directement utilisées par les plateformes telles que Voyager [Recursion Software, 2011], JADE [Bellifemine et al., 2007] et Aglets [Lange et Mitsuru, 1998] pour mettre en œuvre la migration d'agents mobiles. Notons que les références spécifient la date d'apparition de documents que nous avons utilisés pour présenter les plateformes et non à la date de leur première apparition. À titre d'exemple, la plateforme Voyager a été introduite par la compagnie *ObjectSpace* en 1997, achetée depuis par *Recursion Software* mais la version utilisée dans cette présentation date de 2011.

La sérialisation consiste simplement à transférer une instance d'objet en une série d'octets. Cette technique est souvent utilisée pour stocker sur disque une instance d'objet afin d'assurer sa persistance. Cependant, on peut aussi bien transférer cette série d'octets sur le réseau entre deux machines virtuelles Java. C'est d'ailleurs ce qui se passe lorsque l'on fournit un paramètre en appelant un message sur un objet distant. Le paramètre est d'abord transféré en une série d'octets ("sérialisé"), puis acheminé vers la machine virtuelle Java où se trouve l'objet distant. La suite d'octets représentant l'objet-paramètre est ensuite reconstruite comme objet réel dans la mémoire de la machine virtuelle Java de destination ("dé-sérialisé"), puis ce paramètre est passé à l'objet cible lors de l'appel de la méthode.

Toutefois, étant donné que Java ne fournit pas des services d'accès à la zone d'exécution des données, ces plateformes ne permettent pas le support de la migration forte sans les modifications de la machine virtuelle. Elles fournissent la mobilité faible: un agent reprend son cours d'exécution dès le début quand il arrive à la destination.

Au niveau de l'application, la mobilité est mise en œuvre en appelant une méthode donnée, habituellement appelée *moveTo()*, spécifiant d'une façon ou d'une autre le nœud de destination. La figure 3.12 montre deux fragments de code Java d'un agent. La première repose sur la

mobilité faible: l'appel de la méthode *moveTo()* envoie l'agent vers le nœud *NouveauNœud*, puis, après le transfert, l'exécution démarre la méthode *NouvelleMethode()* telle que spécifiée par le paramètre dans la figure 3.12 (a). Dans le cas d'un système supportant la mobilité forte, l'exécution est relancée à partir de la première instruction après la méthode *moveTo()* telle qu'illustrée à la figure 3.12 (b).

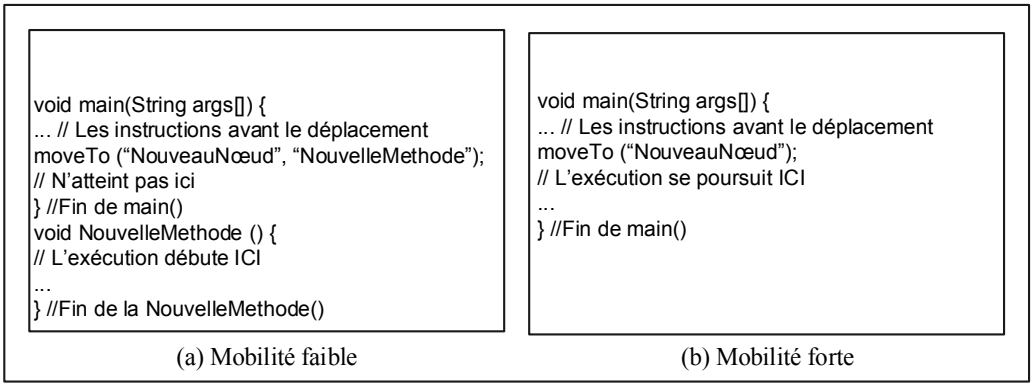


Figure 3.12 Code d'un agent utilisant une mobilité faible en (a) et une mobilité forte en (b)

Nous présentons dans cette section une synthèse des plateformes Voyager, Mobile-C et JADE. Nous nous intéressons particulièrement aux aspects liés à la mobilité des applications puisque ces plateformes sont aussi utilisées dans d'autres domaines (par exemple RPC).

3.4.1 Plateforme Voyager

Cette plateforme a pour objectif de produire rapidement des systèmes répartis. Elle offre une série de services, tous essentiels au développement des applications réparties. Certaines fonctionnalités de la plateforme *Voyager* opèrent en arrière-plan de façon transparente pour l'utilisateur. Les techniques mises en œuvre à l'aide de Voyager permettent la construction des objets distants, de leur envoyer des messages et de les déplacer entres les programmes.

Comme il est difficile de prévoir les exigences uniques de chaque client, la plateforme *Voyager* est construite à base de composants qui peuvent être étendus ou remplacés pour intégrer dans l'infrastructure existante. La plateforme Voyager est mise en œuvre au-dessus de la couche du système d'exploitation et fournit un ensemble d'API universelle [Recursion Software, 2011]. Voyager intègre des composants pouvant s'exécuter aussi bien dans une

machine virtuelle Java que dans un environnement «.net». Ainsi, Voyager devient une plateforme d'abstraction qui permet d'écrire un code natif dans le langage du choix de l'utilisateur (Java, C#, etc.) et de le déployer sur n'importe quel appareil quelle que soit la machine virtuelle du dispositif utilisé. Voyager supporte également plusieurs protocoles de communication tels que SOAP (*Simple Object Access Protocol*), IIOP (*Internet Inter-ORB Protocol*), etc. Voyager est un environnement multi-langage mais c'est seulement le composant écrit en Java qui supporte les agents mobiles et plus précisément la mobilité faible.

3.4.2 Plateforme Mobile-C

Mobile-C [Chou et al., 2010] [Chen et al., 2006] est une plateforme d'agents mobiles qui est mise en œuvre sous la forme d'une bibliothèque. Ainsi, elle peut être facilement intégrée dans des applications qui utilisent des agents mobiles. Cette plateforme est conforme aux normes FIPA et elle est destinée aux systèmes mécatroniques (mécanique/électronique) et aux systèmes embarqués «réseautés». Bien qu'elle soit une plateforme multiagents polyvalente, *Mobile-C* est conçue spécialement pour les systèmes temps réel et pour les machines avec des ressources limitées. Les agents mobiles dans un système de multiagent communiquent et travaillent en collaboration avec d'autres agents pour atteindre un but commun.

Pour des raisons de portabilité, les agents mobiles de *Mobile-C* sont écrits en C/C++. *Mobile-C* est intégré à un interprète appelé Ch [SoftIntegration, 2011] qui est un environnement d'exécution des agents mobiles en C/C++ [Chou et al., 2010]. Ch est un surensemble de C pour l'écriture de script multiplateforme. Cet interprète supporte toutes les caractéristiques de la norme ANSI/ISO de 1990 de C et la plupart des nouveaux éléments ajoutés dans la norme ISO C99**,** tels que les nombres complexes, le tableau de longueur variable, les constantes binaires et le codage de virgule flottante de format IEEE 754. Il supporte aussi les classes, les objets et l'encapsulation en C++ pour la programmation à base d'objet.

Mobile-C est entièrement conforme aux normes FIPA tant au niveau de l'agent qu'au niveau de la plateforme. Au niveau de l'agent, la conformité comprend:

- Le langage de communication entre agents,

- Les protocoles d'échange de message,
- Les actes communicatifs: chaque message est considéré comme une action communicative;
- Les langages de représentation des connaissances pour décrire le contenu des messages.

Au niveau de la plateforme, *Mobile-C* fournit un système de gestion du cycle de vie d'agents, un canal permettant la communication entre agents sur le réseau et un répertoire de pages jaunes.

3.4.3 Plateforme JADE

JADE (*Java Agent DEvelopment Framework*) [Bellifemine et al., 2007] est une plateforme à code source ouvert pour la mise en œuvre de systèmes multiagents conforme aux spécifications FIPA. JADE est un intergiciel qui permet de mettre en œuvre un cadre et un environnement de développement d'agents grâce à un ensemble de bibliothèques Java. Cet intergiciel offre un environnement graphique et fournit aux programmeurs des fonctions prêtes à l'emploi et faciles à personnaliser [Filgueiras et al., 2012].

L'intergiciel permet le développement d'applications d'agents intelligents sur des plateformes fixes ou mobiles. La plateforme JADE est composée de conteneurs d'agent pouvant être distribués à travers le réseau. Les agents vivent dans les conteneurs qui sont des processus Java et fournissent tous les services nécessaires pour héberger:

- L'environnement complet d'exécution pour un agent;
- L'exécution concurrente de plusieurs agents non préemptifs. Cependant, il est important de souligner que l'ordonnanceur non préemptif n'arrête pas l'exécution du processus courant (comme les processus Java). Un ordonnanceur préemptif peut interrompre un processus au profit d'un autre plus prioritaire.
- Le contrôle du cycle de vie des agents,
- La communication entre agents.

Il existe un conteneur spécial, appelé le conteneur principal, qui représente le point d'amorçage de la plate-forme: c'est le premier conteneur à être lancé et tous les autres doivent se joindre à lui. Les conteneurs sont identifiés par leur nom. Une plateforme JADE héberge un ensemble d'agents, identifiés de manière unique, pouvant communiquer de manière bidirectionnelle avec les autres agents. Comme la plateforme JADE respecte la norme FIPA, on retrouve les différents éléments: *AMS, DF,* etc. Il est pertinent de préciser que la plateforme JADE supporte la mobilité faible.

3.5 Applications des agents mobiles

Le paradigme agents mobiles est utilisé dans un certain nombre de domaine et nous donnons des exemples d'applications dans cette section.

3.5.1 Informatique omniprésente

Le travail dans [Satoh, 2002] propose une application à base d'agents mobiles qui permet de suivre le déplacement d'utilisateur. Par exemple, si un utilisateur quitte son bureau pour rentrer chez lui, l'application devrait pouvoir se transférer sur son ordinateur portable. Dans cette approche, l'espace est découpé en zones d'influence qui sont dotées de capteurs. Ces capteurs peuvent détecter des tags RFID *(Radio-frequency identification)* qui servent à identifier diverses entités physiques [Romito, 2012]. Plusieurs agents mobiles peuvent être affectés à un tag RFID de sorte que lorsque ce tag est détecté dans une zone d'influence qui contient une machine pouvant héberger des agents, l'ensemble des agents affectés à ce tag migrent sur la machine libre considérée.

Le travail dans [Urra et al., 2009] propose un système de surveillance de zones à l'aide des capteurs en utilisant des réseaux ad hoc véhiculaires (*VANETs* ou *Vehicular Ad hoc Networks*). Le but de ce système est de réduire le nombre de capteurs nécessaires à la couverture d'un espace en exploitant la mobilité de véhicules évoluant dans cet espace. En effet, la couverture d'un espace avec des capteurs fixes implique un nombre de capteurs proportionnel à la superficie de cet espace. Dans l'approche proposée, un ensemble de véhicules évoluent dans l'espace à surveiller et forment un réseau ad hoc. Chaque véhicule est muni d'un ensemble de

capteurs, d'un GPS ainsi que d'une plateforme agents mobiles. Lorsqu'une zone de l'espace doit être analysée, des agents sont envoyés en direction de cette zone en migrant de véhicule en véhicule dans le but de collecter un maximum d'informations sur cette zone. La politique de déplacement est effectuée à l'aide d'une heuristique sur une vision du voisinage de l'agent. Un des résultats de cette étude stipule que l'augmentation du nombre de véhicules dans la carte contribue à stabiliser les agents de surveillance sur les zones, résultant en un plus grand nombre de relevés [Romito, 2012]. Dans cette section, les auteurs ne mentionnent pas le type de mobilité utilisée. Toutefois, nous pouvons déduire qu'il s'agit d'une mobilité forte. En effet, l'exécution de l'agent est interrompue par un événement externe (un utilisateur quitte, un déplacement d'un véhicule). Lorsque l'agent arrive dans sa machine d'accueil, son exécution reprend au point où elle a été interrompue.

3.5.2 Diagnostic et réparation

Le travail dans [Watanabe et al., 2004] propose un modèle d'agents mobiles de diagnostic et de réparation pour systèmes distribués. Le modèle proposé s'est inspiré du système immunitaire. Le tableau 3.1 présente l'analogie entre le système immunitaire et le système de diagnostic. Le rôle du système immunitaire est de détecter et d'éliminer les substances étrangères telles que les virus et les cellules cancéreuses appelées antigènes. Lorsque l'anticorps reconnaît les antigènes, il neutralise la toxicité d'antigènes par une réaction chimique. Dans ce contexte, un certain nombre d'anticorps et d'antigènes virtuels modélisés sous la forme d'agents mobiles se déplacent dans le réseau.

Tableau 3.1 Analogie entre le système immunitaire et le système de diagnostic

Système immunitaire	**Système de diagnostic**
Antigène	Unités corrompues
Anticorps	Agent mobile de diagnostic
Réseau idiotypique	Diagnostic mutuel
Circulation	Migration d'agents
Neutralisation	Autoréparation

Pour tester le modèle de diagnostic et de réparation, les auteurs ont utilisé une simulation de réseau d'ordinateurs. Ce réseau est composé de N ordinateurs hôtes qui sont reliés entre eux de façon aléatoire. Chaque ordinateur hôte peut envoyer des agents mobiles aux hôtes voisins

via ses connexions. La topologie de réseau est fixée au démarrage de la simulation. Chaque ordinateur hôte possède les quatre composants suivants: le module de diagnostic, le module de réparation, les données et une chaîne de bits. Pour simplifier la simulation, les auteurs utilisent une chaîne de bits où chaque bit est assigné VRAI ou FAUX. La longueur de la chaîne de bits est définie par L. On suppose que tous les bits d'hôtes valides sont assignés à VRAI tandis que les bits corrompus à FAUX. L'hôte corrompu peut découvrir les bits corrompus non par lui-même mais seulement par des comparaisons avec d'autres unités valides.

Au début de la simulation, chaque hôte produit plusieurs agents mobiles ayant une partie des données de l'hôte, le module de diagnostic, le module de réparation et une chaîne de bits. Les hôtes valides créent donc des agents mobiles valides tandis que les hôtes corrompus créent des agents mobiles corrompus ayant des bits erronés. Notons que non seulement les hôtes corrompus sont des cibles pour l'autoréparation mais aussi pour les agents mobiles corrompus. Pendant la simulation, les agents mobiles migrent d'un hôte à l'autre pour le diagnostic et la réparation des hôtes. Il existe également des tests mutuels entre les agents sur le même hôte mais il n'existe aucun test direct entre les hôtes parce que les agents mobiles agissent à leurs noms. Lors de ces tests mutuels, les agents comparent leur état et prennent en compte leur crédibilité. Les agents ont une mesure de confiance sur le fait qu'ils sont plus ou moins corrompus. Dans cette section, les auteurs ne mentionnent pas le type de mobilité utilisée. Toutefois, nous pouvons déduire qu'il s'agit d'une mobilité faible puisque aucune contrainte ne force un agent de migrer avec son contexte d'exécution.

3.5.3 Recherche d'information

Le travail dans [Papadakis et al., 2008] propose une architecture configurable de moteur de recherche à base d'agents mobiles. Dans le modèle de recherche traditionnel centralisé, les données de la source entière doivent être transférées vers le site du moteur de recherche. Dans le modèle distribué proposé par [Papadakis et al., 2008], le moteur de recherche (ou du moins une partie) est transféré vers les données. Puis, la recherche est effectuée localement et les données résultantes sont alors transférées vers le site du moteur de recherche. De manière générale, la taille en octets du moteur de recherche est négligeable par rapport à celle de la

collecte de données, ce qui permet une économie substantielle de bande passante. Par conséquent, l'adoption de l'approche par agent mobile augmente l'efficacité du système. Dans cette section, les auteurs ne mentionnent pas le type de mobilité utilisée. Toutefois, nous pouvons déduire qu'il s'agit d'une mobilité faible. En effet, la recherche d'information dans ces travaux n'implique que deux sites et à la fin de la recherche, les données résultantes sont transférées vers le site de départ.

3.5.4 Commerce électronique

Les agents mobiles sont bien adaptés pour le commerce électronique [Lange et Oshima, 1999]. Une transaction commerciale peut exiger un accès en temps réel à des ressources distantes comme les cours de la bourse et peut même être négociée d'agent à agent. Dans ce contexte, les différents agents ont des objectifs divers et mettent en œuvre leurs stratégies pour les atteindre. Nous envisageons, dans le cas présent, des agents qui représentent les intentions de leurs propriétaires, d'agir et négocier en leur nom. La technologie d'agents mobiles est une solution très intéressante pour ce champ d'activité. Dans cette section, les auteurs ne mentionnent pas le type de mobilité utilisée. Toutefois, nous pouvons déduire qu'il s'agit d'une mobilité faible puisque aucune contrainte ne force un agent de migrer avec son contexte d'exécution.

3.6 Apports et limites du paradigme d'agents mobiles

Certes les agents mobiles offrent des avantages mais ils introduisent aussi des difficultés par rapport aux méthodes de programmation plus classiques.

3.6.1 Avantages et inconvénients

Les agents mobiles apportent de gain de performance due à une meilleure utilisation des ressources physiques mises à leur disposition.

Réduction du trafic réseau

La mobilité des agents permet de réduire significativement les communications distantes entre les clients et les serveurs. En privilégiant les interactions locales, l'utilisation du réseau va se limiter principalement au transfert des agents. Selon Cubat [Cubat dit cros, 2005], cette approche apporte trois principaux avantages.

Le premier avantage est la diminution de la consommation de bande passante. En effet, plusieurs études comparatives [Gray et al., 2002] [Sahai et Morin, 1998a] montrent qu'à la différence du système client-serveur, les agents mobiles peuvent réduire significativement la charge du réseau en terme du nombre total de données transférées. Cette diminution est constatée dans les applications nécessitant des échanges intensifs d'informations entre le client et le serveur. La collecte d'informations dans des bases de données réparties, l'exploration d'internet ou encore la gestion de réseaux sont des exemples de champs d'applications d'agents mobiles.

La diminution de latence est le second avantage [Jun et al., 2002] [Kotz et Gray, 1999] [Johansen, 1998]. Dans le contexte des réseaux à large échelle, la mise en place de ces applications nécessitant des échanges intensifs d'informations entre le client et le serveur se heurte à la latence propre aux communications de réseaux. Il arrive fréquemment que le temps d'attente de la réponse d'une requête soit plus long que le temps de traitement nécessaire à la réalisation du service. En déportant les traitements au plus près des données, on évite la latence de réseau.

Enfin en réduisant le plus possible les communications distantes aux seuls transferts d'agents mobiles, on diminue considérablement les périodes de connexion entre les sites. Cette diminution d'utilisation du réseau nous préserve ainsi des ruptures de communications qui peuvent intervenir particulièrement dans les environnements sans fil.

Calculs indépendants

Dans le modèle classique d'évaluation à distance, le client et le serveur doivent rester connectés tant que le service est en cours d'exécution. Certains services nécessitant de longues phases de traitement ne supportent pas facilement une rupture de connexion avec le client. Dans ce cas, ils doivent souvent redémarrer entièrement leurs calculs. Toutefois, le maintien du lien de communication peut s'avérer difficile dans des réseaux à large échelle ou sans fil. Avec les agents mobiles, un client peut déléguer les interactions avec le service sans maintenir une connexion de bout à bout [Gray et al., 2001].

Tolérance aux fautes physiques

En se déplaçant avec leur code et données propres, les agents mobiles peuvent surmonter les erreurs systèmes. Ces dernières peuvent être d'ordre purement physique, la disparition d'un nœud par exemple, ou d'ordre plus fonctionnel comme l'arrêt d'un service. À titre d'exemple, si un site perd une partie de ses fonctionnalités, l'agent pourra se déplacer vers un autre site offrant la fonctionnalité désirée. Ceci permet une meilleure tolérance aux fautes.

3.6.2 Conception

Les agents mobiles représentent à la fois une méthode permettant de mieux caractériser certaines applications mais ils introduisent aussi une complexité accrue par rapport au modèle classique client-serveur. Avec ce dernier modèle, il est très contraignant de décrire des algorithmes d'exploration (de réseau) ou bien encore de caractériser les déplacements des utilisateurs nomades. Grâce aux paradigmes d'agents mobiles, les concepteurs peuvent décrire naturellement ce type d'application. Ainsi, on peut facilement mettre en œuvre une application permettant la maintenance de réseau [Picco, 1998] ou encore suivre des utilisateurs [Satoh, 2002].

En outre, les agents possèdent une capacité leur permettant de s'adapter à leur environnement. Généralement, dans le modèle client-serveur, le service est caractérisé préalablement par une interface stricte et/ou avec un protocole d'utilisation bien défini. Ainsi, si le client n'a pas une

connaissance de la description du service, il ne pourra pas l'utiliser. Par contre, les agents pourront s'adapter aux caractéristiques des services afin d'exprimer leur requête. Par exemple, si un service demande une communication sécurisée, l'agent pourra récupérer un module de sécurité, mettre à jour sa pile de communication et dialoguer avec le service [Kawaguchi, 2000]. Dans le cas des terminaux avec des ressources limités, l'agent pourra se séparer d'une partie de ses fonctionnalités pour s'adapter à ces environnements [Libsie et Kosch, 2002].

La conception orientée agent permet de mettre en œuvre des agents qui seront autonomes, adaptant leurs déplacements en fonction de l'environnement et pouvant moduler leurs fonctionnalités en cours d'exécution. Cependant, ces propriétés introduisent de nouvelles difficultés qui n'apparaissaient pas dans le modèle classique (client/serveur) [Vigna, 2004]. En effet, dans la majorité des applications distribuées, les environnements d'exécution essaient le plus possible de cacher la répartition en s'appuyant sur un ensemble de services système qui, comme le cas de CORBA [Geib et al., 1999], permettent de concevoir une application comme si tous ses éléments étaient locaux. Pour pleinement tirer parti des agents mobiles, la distribution doit être explicite et gérée par les agents, ce qui complique non seulement la tâche des concepteurs mais également viole le principe de la transparence. En outre, si on se réfère à la conception orientée objet où une application est définie comme un ensemble d'éléments et de relations, il est difficile d'identifier clairement la tâche de chaque agent et surtout de savoir comment et où les interactions vont s'opérer [Vigna, 2004].

3.6.3 Développement

Les applications à base d'agents mobiles se heurtent à différents obstacles lors des phases de développement [Cubat dit cros, 2005]. Le premier obstacle reste le fait qu'il existe un trop grand nombre de plateformes d'agents mobiles possédant chacun leurs propres défauts et qualités [Johansen, 2004]. Avec cette vaste gamme de plateformes, il est difficile de parler de standardisation et aucune d'entre elles ne semble encore s'imposer. Ce manque de standardisation représente un handicap sérieux car les agents ont besoin d'exprimer les

caractéristiques de services qu'ils recherchent et surtout d'obtenir des réponses précises sur leur localisation ainsi que sur la manière de les utiliser.

Le second obstacle important lors des phases de développement est la complexité de déverminer (déboguer) des programmes se déplaçant d'une machine à une autre. La plupart des environnements de programmation possèdent des outils permettant de suivre les différents éléments d'une application durant son exécution afin d'en trouver les erreurs. Cependant, ce genre d'outil est parfaitement adapté aux éléments statiques mais il est difficilement applicable dès que les éléments se déplacent. Faute d'outils de déverminage adéquat, il est très difficile d'observer l'état d'un programme en cours d'exécution durant son déplacement. Notons que des efforts pour proposer un premier outil de déverminage (débogage) sont réalisés sur la plateforme JADE [Bellifemine et al., 2007].

3.6.4 Sécurité

L'utilisation des agents mobiles soulève plusieurs questions dans le domaine de la sécurité et elle comporte trois aspects principaux:

- La confidentialité qui consiste à assurer qu'une information n'est accessible qu'aux entités autorisées à consulter;
- L'intégrité: les données doivent être celles que l'on s'attend à ce qu'elles soient et ne doivent pas être altérées de façon fortuite ou volontaire;
- La disponibilité du service qui correspond au fait que tout sujet autorisé peut avoir accès au service. La non-délivrance d'un service, suite à une attaque, peut être vue comme un trou de sécurité. Cette propriété est également associée aux propriétés de sûreté de fonctionnement.

La mise en œuvre des propriétés de sécurité [Bidan et Issarny, 1995] s'appuie sur des mécanismes d'authentification et de contrôle d'accès [McLean, 1994]. L'authentification permet de certifier l'identité d'un sujet. Dans les systèmes répartis client-serveur, les problèmes de sécurité ont relativement des solutions largement employées. Afin de garantir

aux mieux les propriétés de confidentialité et d'intégrité des objets, les données et les ressources critiques sont localisées sur des machines sécurisées.

Dans les systèmes répartis à base d'agents mobiles, les problèmes de sécurité sont plus délicats. Cette propriété est souvent citée comme la raison majeure qui a limité la diffusion de cette nouvelle architecture. En effet, les agents mobiles posent des nouveaux problèmes de confidentialité, d'intégrité et de disponibilité. Ces problèmes peuvent se diviser en deux catégories. D'une part, le besoin de protéger les sites d'accueil d'agents contre des codes malveillants et d'autre part la protection des agents envers les hôtes mal intentionnés.

Protection d'un site face à un agent mobile

Un système d'exécution devient vulnérable dès qu'il exécute un code venant de l'extérieur s'il n'est pas protégé. Un agent peut facilement violer les propriétés de confidentialité ou d'intégrité s'il a accès à la mémoire ou aux fichiers de la machine hôte. L'agent peut bloquer le système en réalisant des opérations non conformes, consommer de manière illimitée et incontrôlée des ressources, se cloner indéfiniment ou bien migrer sans fin.

Un premier moyen de contrer ces problèmes de sécurité consiste à s'appuyer sur une authentification de l'agent avant son exécution. L'environnement d'exécution est capable de vérifier le niveau de confiance en fonction de la provenance de l'agent. Il peut limiter les droits d'accès aux ressources et données locales. Toutefois, même un code venant d'un site supposé sûr, peut contenir des erreurs non intentionnelles qui peuvent se répercuter sur la sécurité du système hôte. Trois techniques peuvent être employées pour contrer les malveillants:

- Les interprètes sûrs représentent l'approche la plus largement utilisée pour résoudre le problème du code malveillant. Une machine virtuelle permet l'utilisation d'un gestionnaire de sécurité qui vérifie chacune des instructions et choisit de les exécuter ou non. Plus particulièrement, la machine virtuelle Java utilise une vérification statique du code et un gestionnaire de sécurité afin de contrôler les instructions du *bytecode* Java. Les accès aux ressources critiques sont ainsi contrôlés à la manière d'un pare-feu local.

- La technique de l'isolation permet de charger le code non sûr et de s'exécuter dans un espace d'adresse restreint (c'est-à-dire bac à sable) [Wahbe et al., 1993]. Le site hôte vérifie le code reçu au moment du chargement. Puis, il le sauvegarde ou les instructions de sauts se font dans une adresse du bac à sable. Ainsi, les accès mémoire sont confinés dans un espace et les données privées du site sont protégées. Les appels systèmes sont traités à part via une table de sauts. Pour illustrer cette technique, nous pouvons citer les applets Java exécutés à l'intérieur d'un navigateur Web.
- La vérification automatique du code de l'agent sur le site hôte avant son exécution s'appuie sur la vérification d'une preuve de conformité. Necula et Lee [Necula et Lee, 1998] ont proposé une approche où le propriétaire (la machine source) de l'agent doit produire une preuve formelle que son agent garantit des propriétés de sécurité définies par un site hôte. Pour ce faire, les auteurs ont utilisé l'approche PCC (Proof Carrying Code) [Necula et Lee, 1998]. Le code et la preuve sont envoyés simultanément sur le site hôte. L'hôte vérifie que la preuve est formellement correcte et exécute l'agent si et seulement si il est en accord avec la politique. L'approche PCC utilise un ensemble d'axiomes et de règles de réécriture suivant la logique choisie et partagée par la machine source du code et la machine destinateur.

Protection d'un agent mobile face à un site hôte

Les environnements d'exécution des agents sur les sites doivent avoir accès au code et à l'état des agents afin de pouvoir les exécuter. Durant l'exécution sur un serveur, le code, l'état et les données de l'agent sont exposés à des problèmes de confidentialité et d'intégrité. Les attaques possibles ne sont pas encore toutes identifiées [Hohl, 1998]. En ce qui concerne la confidentialité, un site malveillant peut accéder au programme d'un agent afin de découvrir des informations privées transportées par l'agent. L'étude du flot de contrôle de l'agent ou de son itinéraire peut permettre de découvrir de l'information sur des états antérieurs de l'agent. Un site peut exploiter ces informations pour modifier le comportement de ses services.

À propos de l'intégrité, il y a deux cas à considérer:

- Le site hôte peut rejouer l'exécution, cloner ou incorrectement exécuter le code de l'agent. Il peut aussi fausser les résultats des appels systèmes faits par l'agent et ainsi duper l'agent.
- Le site hôte peut modifier le code, l'état ou les données d'un agent. Ces modifications peuvent générer des actions malveillantes envers des hôtes que l'agent visitera ultérieurement afin de monter une attaque. Si une plateforme réceptrice ne peut pas déterminer qu'un agent accueilli a été falsifié, il convient alors de le traiter comme un agent anonyme et lui donner des permissions minimales.

La propriété de disponibilité peut être affectée en termes de refus d'exécution. L'environnement d'accueil d'un agent peut par exemple décider à tout moment de terminer brutalement l'exécution de l'agent ou empêcher sa migration.

La solution la plus triviale pour le propriétaire d'un agent est de limiter son itinéraire à des sites de confiance. La seule défense réelle concrète des agents contre des sites malveillants provient du domaine du matériel. Cette approche [Wilhelm et al., 1999] [Wilhelm et al., 1998] impose l'exécution d'agents exclusivement sur du matériel prouvé bienveillant par un tiers. Un tel composant matériel, connecté à un service offert sur le réseau et régulièrement vérifié par le tiers, dispose d'un environnement d'exécution.

Dans les approches logicielles, la protection des agents contre des sites malveillants ne dispose pas de solution complète et reste encore un champ de recherche ouvert. Il existe cependant des solutions partielles qui traitent des sites mal intentionnés afin de garantir certaines propriétés de sécurité. Young et Yung [Young et Yung, 1997] ont chiffré les données d'agents accumulées et non-réutilisées durant le déplacement. L'agent peut alors traverser des sites douteux en gardant la confidentialité/intégrité des petites données accumulées. La technique se base sur un chiffrement à clé publique des données sensibles. L'algorithme de chiffrement et la clé sont définis dans le code de l'agent. La donnée chiffrée ne peut être déchiffrée que sur le site du client (source de l'agent) qui possède la clé privée. La clé privée ne circule pas avec l'agent.

3.6.5 Fiabilité

Le domaine de la sûreté de fonctionnement est assez vaste et intervient dans le matériel, le logiciel ou dans des systèmes répartis. La sûreté de fonctionnement d'un système est la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il leur délivre [Laprie, 1995]. Une application répartie sûre de fonctionnement doit pouvoir continuer de fonctionner en présence de défaillance dans le système. Une défaillance est une déviation du comportement d'un composant envers sa spécification [Laprie, 1995]. Une défaillance se produit à la suite d'une erreur (état incorrect du système). Cette erreur peut être induite par un phénomène extérieur (par exemple un phénomène physique) ou une imperfection dans le système (par exemple une faute de conception).

Dans une architecture répartie, les défaillances de site peuvent conduire à une défaillance de système et donc le rendre inutilisable au client (par exemple le client ne récupère pas ses résultats ou ils sont incorrects). Plusieurs types de défaillances peuvent être considérés [Chevochot et Puaut, 1997]:

- une défaillance par arrêt (par exemple une panne) quand un serveur ne rend pas de résultats suite à des appels répétés;
- une défaillance par omission quand un serveur omet de répondre à son client;
- une défaillance temporelle quand la réponse du serveur est fonctionnellement correcte mais n'est pas arrivée dans un intervalle de temps donné;
- une défaillance de valeur quand le serveur rend des résultats incorrects.

3.6.6 Évaluation des performances

Dans cette section, nous allons présenter les évaluations de performances entre le paradigme d'agents mobiles et le modèle traditionnel client-serveur dans deux différents domaines d'application: la recherche d'information et la gestion de réseau.

Recherche d'information

Une des applications d'agents mobiles la plus importante est la recherche d'information. Dans ces applications, des agents se déplacent sur différents sites pour chercher des informations pour leurs clients. Dans [Ismail et Hagimont, 1999], les auteurs ont réalisé une application répartie sur Internet dont le but était de chercher une liste d'hôtels dans une ville. Cette application consiste à consulter deux bases de données. La première recense des hôtels et permet d'obtenir la liste des hôtels de la ville où le client désire se rendre. La deuxième base de données est un serveur annuaire permettant d'obtenir les numéros de téléphone de ces hôtels. Ces deux bases de données sont gérées sur des sites différents par des administrations ou des entreprises différentes.

D'abord, en utilisant le modèle client-serveur classique, le client va devoir faire un appel à distance (en utilisant par exemple le mecanisme d'appel de procédure à distance, en anglais *Remote Procedure Call, RPC*) pour interroger le premier serveur *(serveur A)* et récupérer la table des hôtels qui l'intéressent. À partir de cette table, le client va ensuite, pour chaque hôtel dans la table, réaliser un appel à distance au deuxième serveur *(serveur B)* afin de récupérer le numéro de téléphone de l'hôtel.

Dans le mode client-serveur (voir la figure 3.13 (a)), le client doit appeler le second (serveur B) autant de fois qu'il y a d'éléments dans le tableau d'hôtels retourné par le premier (serveur A). Le temps d'obtention de la réponse finale est égal au coût de communication avec le serveur A (rpc(A)), plus le coût des communications avec le serveur B qui est égal au coût de l'appel au serveur B multiplié par la taille du tableau obtenu comme réponse du serveur A (n*rpc(B)). Le coût total du modèle client et les serveurs A et B peut être calculé [Ismail et Hagimont, 1999]:

coût total = rpc(A) + n * rpc(B)

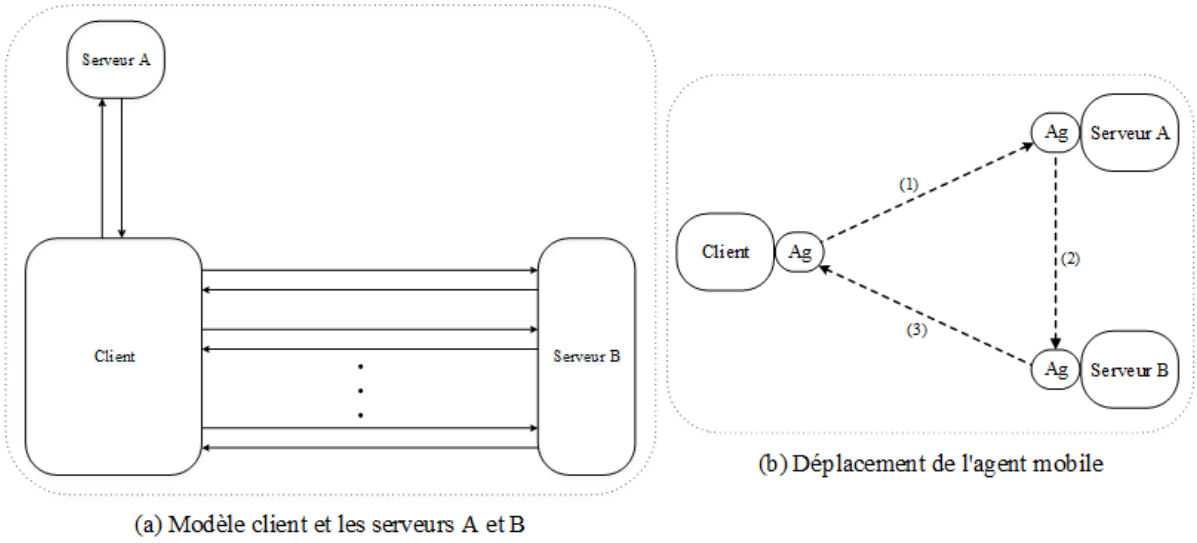


Figure 3.13 Paradigme d'agents mobiles versus le modèle client-serveur

Une autre méthode consiste à transmettre directement la table des hôtels du serveur A au serveur B afin que celui-ci réalise une jointure avec sa table des numéros de téléphone. Pour ce faire, il faut spécialiser les serveurs pour qu'ils offrent ce service de redirection de requêtes. Une extension statique des serveurs (par les administrateurs des deux serveurs) ne constitue pas une solution réaliste dans la mesure où ces bases de données sont administrées séparément. De plus, il n'est pas réaliste d'étendre un serveur pour chaque besoin spécifique d'un client.

Une approche par des agents mobiles est donc proposée comme illustré à la figure 3.13 (b) [Ismail et Hagimont, 1999]. Le client crée un agent mobile appelé *Ag* contenant la requête globale à réaliser. L'agent *Ag* se déplace (1) tout d'abord vers le *serveur A* et réalise localement l'appel pour obtenir la table des hôtels. La table des hôtels est stockée dans le contexte de l'agent, puis l'agent se déplace (2) vers le *serveur B*. Sur ce dernier, l'agent peut réitérer sur la table des hôtels et demander le numéro de téléphone de chaque hôtel. Ces numéros de téléphone sont stockés dans le contexte de l'agent qui se déplace (3) finalement vers son site d'origine où il délivre le résultat au client.

Dans cette approche, un agent doit être envoyé sur le serveur A *(coût A)*, puis il se déplace avec la table des hôtels vers le serveur B (*coût A(TableHotel)*) et retourne à son site d'origine

avec la table contenant les numéros de téléphone *(coût A(TableTelephone))*. Le coût total du modèle d'agent mobile peut être calculé [Ismail et Hagimont, 1999]:

coût *total = A + A(TableHotel) + A(TableTelephone)*

Ainsi, l'approche à base d'agents mobiles sera plus efficace si [Ismail et Hagimont, 1999]:

*A + A(TableHotel) + A(TableTelephone) < rpc(A) + n * rpc(B)*

Des mesures ont été effectuées en variant le nombre d'enregistrements retournés par le premier serveur. Dans cette application, la taille d'un enregistrement est de 80 octets. La figure 3.14 [Ismail et Hagimont, 1999] illustre les résultats de l'expérience.

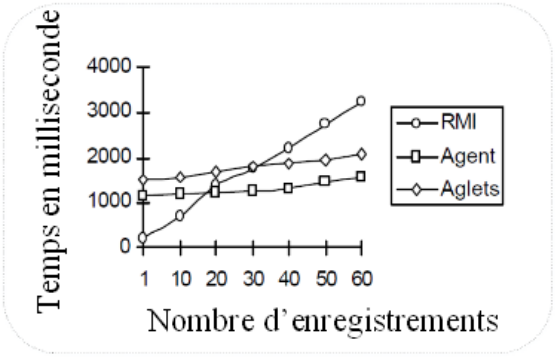


Figure 3.14 Comparaison entre RMI et les agents mobiles pour l'application [Ismail et Hagimont, 1999]

On observe que pour un nombre d'enregistrements petit (moins de 20 enregistrements dans le cas présent), la mise en œuvre basée sur RMI (pour *Remote Method Invocation*) est plus efficace que celle basée sur des agents mobiles. Ceci s'explique par le fait que pour un nombre d'enregistrements petit, le nombre d'appel à distance économisé n'est pas suffisant pour amortir le coût de la migration de l'agent sur le réseau. Cependant, pour un nombre d'enregistrements suffisant, l'approche à base d'agent est bien plus efficace.

La différence entre *Aglets* et *Agent* [Ismail et Hagimont, 1999] s'explique par le fait que *Agent* est un prototype minimal n'implantant que les fonctions strictement nécessaires. Dans ce travail [Ismail et Hagimont, 1999], les auteurs précisent qu'il s'agit d'une mobilité forte.

Gestionnaire de réseau mobile

Avant de présenter le gestionnaire de réseau mobile, nous allons d'abord faire un rappel du protocole SNMP qui est traditionnellement utilisé pour l'administration distante de réseau. Le protocole SNMP est basé sur le modèle client-serveur. L'architecture de gestion du réseau proposée par le protocole SNMP est fondée sur les trois principaux éléments [RFC 3411, 2002] [FrameIP TcpIP]:

1. Les équipements gérés (*managed devices*) sont des éléments du réseau contenant des objets de gestion (*managed objects*) pouvant être des informations sur le matériel, des éléments de configuration ou des informations statiques.
2. Les agents SNMP, c'est-à-dire les applications de gestion de réseau résidant dans un périphériques, sont chargés de transmettre les données locales de gestion du périphérique au format SNMP.
3. Les systèmes de gestion de réseau (*network management systems*) sont les stations à travers lesquelles les administrateurs peuvent réaliser des tâches d'administration.

Les stations d'administration sont les machines qui centralisent toutes les données. En effet, la station est le cœur du système et c'est elle qui va dialoguer avec les différents matériels à administrer. Les agents SNMP, qui sont des entités logicielles, se trouvent au niveau de chaque interface connectant au réseau l'équipement géré (nœud) et permettent de récupérer des informations sur différents objets.

Les commutateurs, les concentrateurs, les routeurs, les postes de travail et les serveurs sont des exemples d'équipement contenant des objets gérables. Ces derniers peuvent être des informations matérielles, des paramètres de configuration, des statistiques de performances et autres objets qui sont directement liés au comportement en cours de l'équipement en question. Ces objets sont classés dans une sorte de base de données arborescente appelée *MIB (Management Information Base MIB)*. L'arbre *MIB* a comme feuille des variables SNMP. Ces dernières sont identifiées par des *OID (Object Identifier)*. Le protocole SNMP est utilisé pour connaître à un instant donné l'état d'un matériel. Ainsi, ce sont ces *variables SNMP* qui permettront de connaître par exemple le nombre de paquets entrants et sortants sur une

interface donnée ou encore la température du processeur d'un serveur, etc. La version 3 du protocole SNMP est le standard actuel et il est défini dans [RFC 3411, 2002].

Une fois décrite le protocole SNMP, revenons à la présentation de la gestionnaire de réseau mobile. Cette dernière comprend essentiellement la surveillance et le contrôle des éléments de réseau. Un gestionnaire de réseau mobile (en anglais Mobile Network Manager, MNM) est un administrateur de réseau via un ordinateur portable [Sahai et Morin, 1998a] [Sahai et Morin, 1998]. Le réseau comprend des éléments fixes exécutant des agents SNMP (pour Simple Network Management Protocol). Le MNM interroge les agents SNMP sur les variables de gestion de réseau et obtient leurs valeurs. Les agents sont utilisés pour étudier la performance des composants de réseau, installer le logiciel, vérifier les composants de réseau (par la collecte de l'information sur le disque, les utilisateurs de la machine, la configuration de l'application, etc.) et la découverte de réseau.

Le MNM opère en mode filaire ou sans fil. Le MNM envoie typiquement un grand nombre d'agents mobiles pour mettre en œuvre une série d'actions désirées. Après avoir terminé son parcours, l'agent mobile peut se déconnecter du réseau et rester dans sa dernière place (machine) de son itinéraire. Le MNM et les agents SNMP sont intégrés à des places pour qu'ils puissent transférer et accueillir les agents mobiles. Dans le cas où le MNM disparait brusquement, l'agent essaie d'atteindre la place MNM pour se connecter. Lorsque le MNM se connecte de nouveau, toutes les places sont informées de l'apparition de la place MNM. Ainsi, l'agent en attendant la place MNM revient avec les résultats.

Pour évaluer les performances, deux MNMs ont été donc mis en place, l'une utilisant des agents mobiles et l'autre le mécanisme client-serveur. La figure 3.15 (a) présente la comparaison de l'utilisation de bande passante lorsque le client-serveur et l'agent mobile sont utilisés pour obtenir un nombre donné d'échantillons d'une seule variable d'un agent SNMP. Comme nous le montre la courbe de la figure 3.15 (a), lorsqu'on désire un grand nombre d'échantillons, en termes de consommation de bande passante, il est préférable d'utiliser le paradigme d'agents mobiles. La figure 3.15 (b) montre une comparaison similaire à la précédente mais celle-ci est obtenue avec des échantillons de 5 variables SNMP.

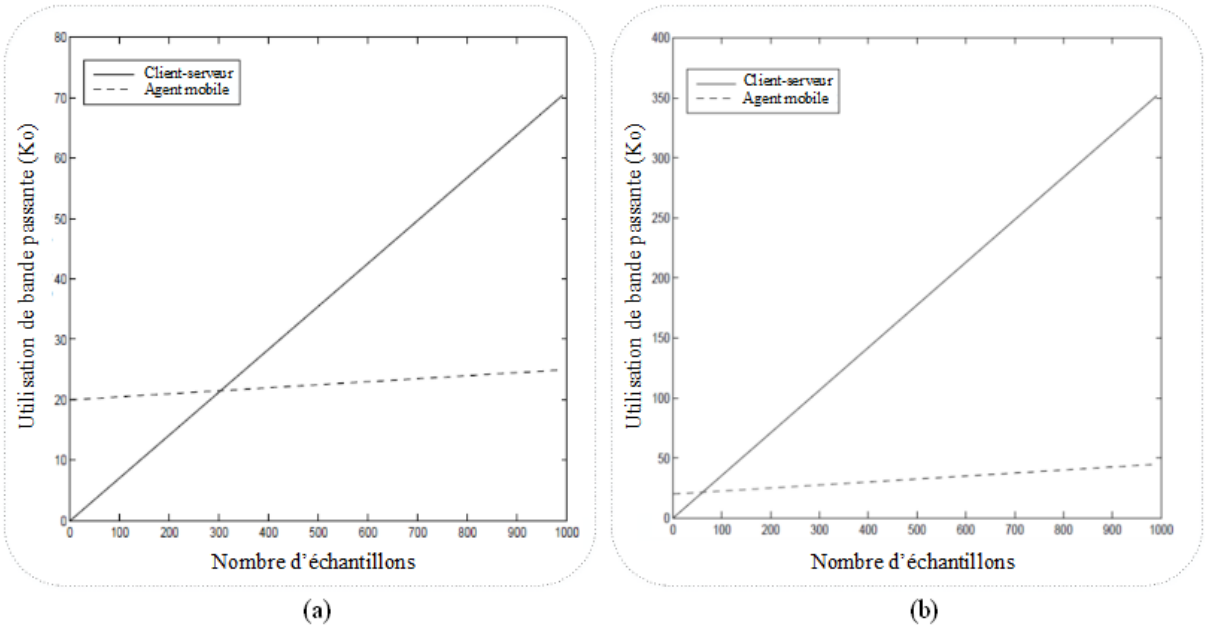


Figure 3.15 Courbes de l'utilisation de bandes passantes du paradigme d'agents mobiles versus le mécanisme client-serveur [Sahai et Morin, 1998a]

La figure 3.16 illustre une comparaison des temps de réponse entre le client-serveur et l'agent mobile. Les courbes de la figure 3.16 illustrent qu'il est plus efficace d'utiliser le paradigme d'agents mobiles lorsqu'on a un plus grand nombre d'échantillons de l'agent SNMP à interroger. Dans ce travail [Sahai et Morin, 1998a], les auteurs précisent qu'il s'agit d'une mobilité forte.

Le MNM a été exécuté sur [Sahai et Morin, 1998a]:

- Un ordinateur portable IBM ThinkPad 760 EL
- Un ordinateur de bureau Pentium 133 MHz ayant une mémoire de 16 Mo
- Un système d'exploitation Windows 95
- Une carte réseau Ethernet de 10 Mbps.

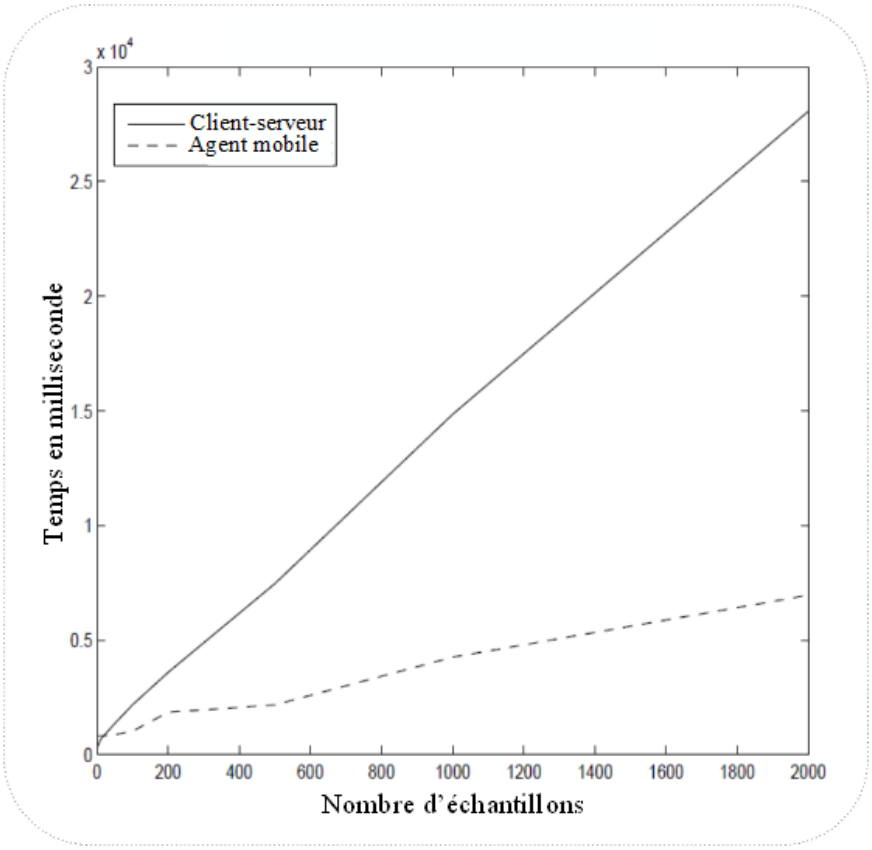


Figure 3.16 Comparaison des temps de réponse entre le client-serveur et l'agent mobile [Sahai et Morin, 1998a]

3.7 Conclusion et positionnement de la thèse

Après avoir consacré le deuxième chapitre à la présentation de la migration des processus dans des machines homogènes et hétérogènes, dans ce chapitre, nous nous sommes intéressés au domaine des agents mobiles. Nous avons commencé par définir le concept d'agent. Ensuite, nous avons présenté les efforts de standardisation des plateformes d'agents mobiles, à savoir les normes MASIF et FIPA, pour promouvoir l'interopérabilité. Les deux normes offrent une description textuelle des concepts de:

- Système d'agents, la place, la région, etc. dans le cas MASIF;
- Système de gestion d'agents, le service de transport de messages, la plateforme d'agents, etc. dans le cas FIPA.

Les deux normes offrent aussi des représentations graphiques pour illustrer les liens entre ces concepts.

Nous avons aussi présenté les travaux sur la modélisation des systèmes à base d'agents mobiles selon deux grandes classes:

- La modélisation structurelle qui permet d'illustrer l'architecture générale d'un système, ses composants ainsi que ses différents types de liens d'interactions;
- La modélisation comportementale qui met l'accent sur l'évolution et l'interaction des agents mobiles dans leur environnement réparti.

Dans ce chapitre, nous avons également décrit des exemples de plateformes telles que Voyager, Mobile-C et JADE pour illustrer la mise en œuvre des concepts d'agents mobiles. Des descriptions détaillées des plateformes d'agents mobiles se trouvent dans l'annexe A. Le paradigme agents mobiles est utilisé dans un certain nombre de domaine dont nous avons présenté les exemples d'applications.

Nous avons aussi présenté des évaluations de performances entre le paradigme d'agents mobiles et le modèle traditionnel client-serveur dans deux différents domaines d'application: la recherche d'information et la gestion de réseau.

Les agents mobiles offrent des avantages mais présentent aussi des difficultés par rapport aux paradigmes de programmation plus classiques. Pour illustrer les avantages offerts par l'utilisation des agents mobiles, nous avons présenté les domaines d'application comme la réduction du trafic réseau, les calculs indépendants et la tolérance aux fautes physiques. Ils permettent aussi de mieux caractériser certaines applications.

Certes la technologie des agents mobiles offre des avantages mais elle vient aussi avec son lot d'inconvénients dont les principaux sont:

- Le manque de standardisation entre le grand nombre de plateformes d'agents mobiles. Malgré les efforts de standardisation de MASIF et FIPA, cette vaste gamme de plateformes ne facilite pas l'interopérabilité d'agents issus de différents organismes. Il en résulte une très grande incompatibilité entre les différentes plateformes [Romito, 2012].
- La complexité de déverminer des programmes se déplaçant d'une machine à une autre;

- La difficulté de sécuriser les plateformes d'agents mobiles: Le plus gros obstacle actuel à l'utilisation de cette technologie [Roth, 2004] demeure le manque d'une protection d'agents mobiles contre les attaques de plateformes malveillantes.

Dans ce chapitre, nous avons également présenté différents types de migration de programme qui peut être classifiés selon deux grandes caractéristiques. La première est la prise de décision de la migration: proactive ou réactive. Lorsque la migration est provoquée par une unité extérieure de l'application, on parle alors d'une migration réactive. En revanche, si la migration est à l'initiative du code de l'application, nous parlons d'une migration proactive et ceci donne de l'autonomie aux agents. Et c'est précisément cette autonomie qui différencie les agents mobiles de la migration de processus. La seconde caractéristique est le degré de mobilité de l'application, à savoir les migrations forte, faible ou du code seulement. Le choix de degré de mobilité dépend du type de l'application mais nous estimons que la migration forte offre plus de capacité aux agents.

Après avoir étudié la migration de processus (chapitre 2) et les agents mobiles (chapitre 3), nous avons constaté qu'il n'existe pas des plateformes d'agents mobiles pour systèmes embarqués temps réel supportant la mobilité forte. Bien que la plateforme *Mobile-C,* par exemple, soit conçue spécialement pour les systèmes embarqués temps réel, celle-ci ne supporte pas la mobilité forte. En outre, la plateforme *Mobile-C* n'est pas destinée pour les systèmes des ressources très limitées. Un programme *Mobile-C* peut prendre un espace de mémoire de l'ordre de 6 Mo [*Mobile-C*, 2011]. Cet espace de mémoire est trop gros pour de nombreux systèmes embarqués. En effet, ces systèmes utilisent un espace de mémoire de l'ordre de centaines de kilooctets seulement.

3.7.1 Solutions existantes de migration et les systèmes embarqués temps réel

Les solutions de migration de processus/agents proposées jusqu'ici ne fonctionnent que sur des machines ayant plus de ressources: un espace mémoire de grande taille, un module matériel pour la gestion de mémoire, etc. La principale caractéristique de ces machines reste

leur fonction de mémorisation. Celle-ci est organisée en plusieurs niveaux de mémoire qui réalisent ainsi une hiérarchie de mémoire dont le but principal consiste à simuler l'existence d'une mémoire rapide de grande capacité. Dans cette hiérarchie, les mémoires les plus proches du processeur sont les plus rapides mais aussi les plus petites en capacité. En revanche, les mémoires les plus éloignées du processeur sont les plus lentes mais aussi les plus grandes en capacité. En effet, les solutions existantes de migration de processus/agents ne sont basées que sur des machines supportant les mécanismes de mémoire virtuelle. Pour déployer un système d'exploitation supportant la mémoire virtuelle comme Windows ou Linux, la machine hôte doit avoir un espace de mémoire disponible de l'ordre des centaines de mégaoctets. De plus, il faut ajouter à cela un espace de mémoire utilisé par une machine virtuelle dans le cas de migration d'agents/processus dans les systèmes hétérogènes.

En somme, les solutions existantes de la mobilité d'agents/processus ne sont pas utilisables dans le contexte des systèmes embarqués en raison de ressources limitées. De manière générale, les systèmes embarqués sont soumis aux contraintes suivantes [Babau, 2005]:

- Le faible espace mémoire disponible;
- Le temps réel lié aux exigences du procédé contrôlé ou aux utilisateurs vis-à-vis des applications supportées;
- L'autonomie limitée en matière d'énergie due à l'utilisation de batterie;
- L'enfouissement: l'absence d'une interface homme-machine ou le manque de connexion à un réseau fixe, etc.;
- L'optimisation du coût de production;
- Les contraintes physiques de l'application (électriques, mécaniques et thermiques);
- Ou encore de nature ergonomique à cause de la taille réduite des systèmes pour assurer leur portabilité ou leur intégration dans un système physique.

L'apport unique et l'originalité du travail accompli dans cette thèse résident dans la conception ainsi que dans la mise en œuvre d'une plateforme d'agents mobiles pour système embarqué temps réel. À titre comparatif, le tableau 3.2 présente une grille d'évaluation entre les caractéristiques principales du système de migration d'agents réalisé dans le cadre de cette

thèse et celles des systèmes existants. Nous avons nommé *µC/MAS (*pour *Microcontroller Mobile Agent System)* notre plateforme d'agents mobiles pour systèmes embarqués.

Comme nous avons déjà décrit, les systèmes de migration de processus/agents au niveau application peuvent offrir, dans certains cas, une migration forte. Cependant, comme ces systèmes ne sont pas en mesure d'accéder à l'état d'exécution complet d'un processus ils peuvent, parfois, ne pas tenir en compte des résultats intermédiaires. C'est le cas du système Wasp qui entraine une migration incomplète. Le Tableau 3.2 tient aussi compte de la complétude de l'état d'exécution.

Tableau 3.2 Comparaison des systèmes traitant la mobilité

Caractéristique / Système	Degré de mobilité	Complétude de l'état d'exécution	Type de mobilité	Empreinte mémoire	Nécessité d'un MMU pour fonctionner	Systèmes embarqués temps réel
µC/MAS	Forte	Oui	Proactive	De l'ordre des dizaines de Ko	Non	Oui
Mobile-C	Faible	Non	Proactive	De l'ordre des dizaines de Mo	Oui	Oui
Tui	Forte	Oui	Proactive	De l'ordre des dizaines de Mo	Oui	Non
CTS	Forte	Oui	Proactive	De l'ordre des centaines de Mo	Oui	Non
ITS	Forte	Oui	Proactive	De l'ordre des centaines de Mo	Oui	Non
Wasp	Faible	Non	Proactive	De l'ordre des centaines de Mo	Oui	Non
JavaGo	Faible	Non	Proactive	De l'ordre des centaines de Mo	Oui	Non
Brakes	Faible	Non	Proactive	De l'ordre des centaines de Mo	Oui	Non
JavaGoX	Faible	Non	Proactive	De l'ordre des centaines de Mo	Oui	Non
MOSIX	Forte	Oui	Réactive	De l'ordre des centaines de Mo	Oui	Non
Charlotte	Forte	Oui	Réactive	De l'ordre des centaines de Mo	Oui	Non
Système V	Forte	Oui	Réactive	De l'ordre des centaines de Mo	Oui	Non
Accent	Forte	Oui	Réactive	De l'ordre des centaines de Mo	Oui	Non
Sprite	Forte	Oui	Réactive	De l'ordre des centaines de Mo	Oui	Non
Choices	Forte	Oui	Réactive	De l'ordre des centaines de Mo	Oui	Non
Smile	Forte	Oui	Réactive	De l'ordre des centaines de Mo	Oui	Non
RHODOS	Forte	Oui	Réactive	De l'ordre des centaines de Mo	Oui	Non
Mach	Forte	Oui	Réactive	De l'ordre des centaines de Mo	Oui	Non
Voyager	Faible	Non	Proactive	De l'ordre des centaines de Mo	Oui	Non
JADE	Faible	Non	Proactive	De l'ordre des centaines de Mo	Oui	Non

CHAPITRE 4
ARCHITECTURE DE LA SOLUTION DE MOBILITÉ D'AGENTS PROPOSÉE

Dans le chapitre précédent, nous avons exposé un état de l'art qui porte sur les caractéristiques, la modélisation, les efforts de normalisations des plateformes, les domaines d'application ainsi que les avantages et inconvénients des agents mobiles. Nous y avons aussi présenté l'étude des travaux effectués dans le domaine de la migration de processus qui est la base des systèmes d'agents mobiles.

Dans le présent chapitre, nous décrivons les différents éléments de notre solution de mobilité qui se traduit par le développement d'une plateforme d'agents mobiles pour systèmes embarqués reposant sur l'extension d'un noyau temps réel. Pour commencer, nous présentons la méthode de migration d'agents. Ensuite, nous exposons la directive de migration d'agents. Enfin, nous décrivons le format de transfert d'agents.

4.1 Introduction

Depuis l'émergence du concept d'agents mobiles, plusieurs plateformes ont été développées pour faciliter la mise en œuvre des applications distribuées. Ces plateformes utilisent presque exclusivement un langage interprété (machine virtuelle). En fait, en raison de sa machine virtuelle disponible sur la très grande majorité des systèmes, les applications à base d'agents mobiles sont écrites presque exclusivement en Java. Ce dernier fournit un mécanisme pour la sérialisation et le chargement dynamique de classes qui sont directement utilisés pour mettre en œuvre les plateformes d'agents mobiles telles que Voyager [Recursion Software, 2011], JADE [Bellifemine et al., 2007], Aglets [Lange et Mitsuru, 1998], pour en nommer quelques-unes. La figure 4.1 présente un modèle conceptuel de plateforme d'agents mobiles basée sur la machine virtuelle Java.

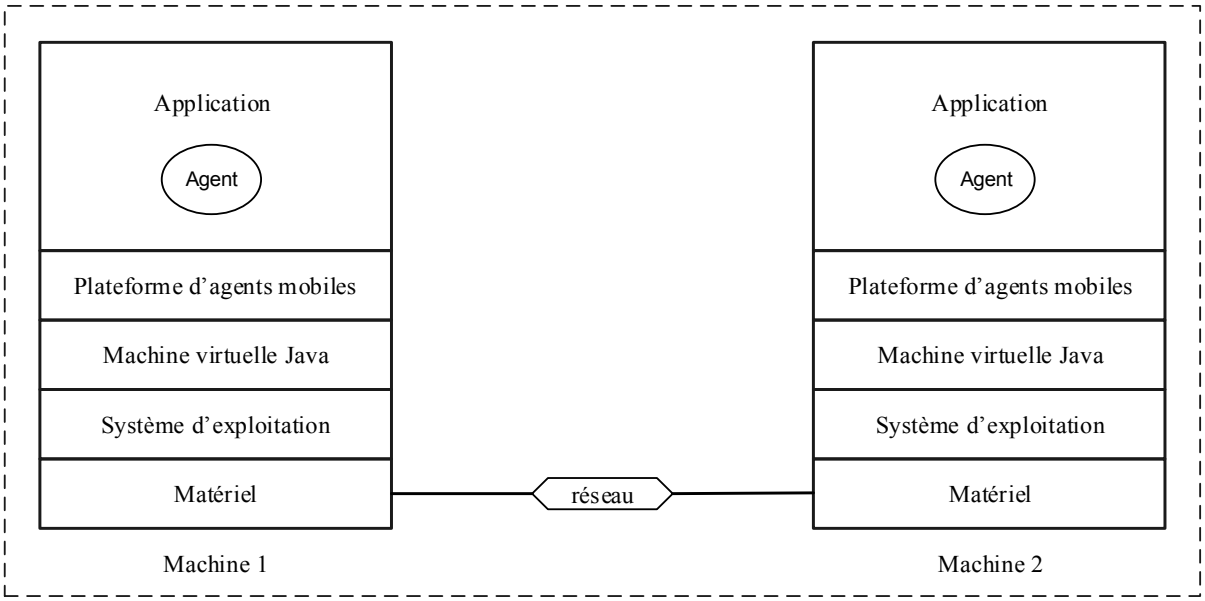


Figure 4.1 Modèle conceptuel d'une plateforme d'agents mobiles basée sur la machine virtuelle Java

Cependant, Java et les autres langages qui utilisent des machines virtuelles sont trop gourmands notamment en termes d'espace mémoire utilisé pour de nombreux systèmes embarqués. Une caractéristique clé de systèmes embarqués est qu'ils opèrent sur des machines avec des ressources limitées. Ces contraintes sont généralement d'ordre spatial (taille limitée) et énergétique (consommation restreinte). Les systèmes embarqués exécutent des tâches prédéfinies et ont des contraintes qui peuvent être liées:

- À l'espace mémoire limité: concevoir des systèmes embarqués qui répondent au besoin strict pour éviter un surcoût.
- À la nécessité d'une puissance de calcul juste pour répondre aux besoins de la tâche prédéfinie en évitant ainsi un surcoût de l'appareil et une consommation excédentaire d'énergie.
- À la consommation énergétique la plus faible possible due à l'utilisation de batteries.
- Aux contraintes matérielles (poids, encombrement, etc.): les systèmes embarqués, dont il est primordial de minimiser la taille et le poids, doivent cohabiter sur une faible surface.
- Aux temps d'exécution d'une tâche qui sont déterminés (les délais sont connus ou bornés a priori). Ces systèmes ont des propriétés temps réel qui se différencient des autres par la prise en compte de contraintes temporelles dont le respect est aussi important que l'exactitude du résultat. Autrement dit, le système temps réel ne doit pas

seulement délivrer des résultats exacts mais il doit aussi les délivrer dans des délais imposés.

- À la sûreté de fonctionnement: Il arrive que certains de ces systèmes embarqués subissent une défaillance, en mettant, par exemple, des vies humaines en danger ou des investissements importants en péril. Ils sont alors dits «critiques» et ne doivent jamais faillir. Par «jamais faillir», il faut comprendre toujours donner des résultats justes, pertinents et ce dans les délais attendus par les utilisateurs (machines et/ou humains).
- À la sécurité: Ces systèmes peuvent se révéler être porteurs d'informations confidentielles pour leur utilisateur. Ces systèmes doivent conserver et protéger, notamment, en ce qui concerne l'acquisition et la transmission d'informations médicales, par exemple.
- À l'environnement: Les systèmes embarqués sont soumis à de nombreuses contraintes dictées par l'environnement telles que la température, l'humidité, les vibrations, les chocs, les variations d'alimentation, les interférences RF, la corrosion, l'eau, le feu, les radiations, etc.

À notre connaissance, nous sommes les premiers à concevoir une plateforme d'agents mobiles pour systèmes embarqués temps réel supportant la migration forte. En effet, les plateformes comme *Voyager* et *Mobile-C* supportent les agents mobiles pour systèmes embarqués, cependant, elles ne supportent que la migration faible: l'agent mobile reprend son exécution depuis le début lorsqu'il arrive à la destination.

En outre, la plateforme *Mobile-C* utilise un espace mémoire de l'ordre de 6 mégaoctets. La plateforme *Voyager*, quant à elle, utilise la machine virtuelle Java (donc dans l'ordre de centaines de mégaoctets). En effet, selon les informations que nous avons obtenues du constructeur, l'espace mémoire requis est de 500 mégaoctets pour déployer la plateforme Voyager. Comme les autres plateformes utilisant la machine virtuelle Java, *Voyager* n'est pas conçue pour les applications temps réel à cause du caractère non déterministe du ramasse-miettes inhérent. En effet, le ramasse-miettes, ou le récupérateur de mémoire, est un composant de la machine virtuelle Java de gestion automatique de la mémoire. Il est responsable du recyclage de la mémoire préalablement allouée puis inutilisée. Le mécanisme

de récupération de mémoire n'est pas déterministe puisqu'il est difficile de borner ce temps d'exécution. L'utilisation d'un ramasse-miettes standard peut donc rendre difficile l'écriture de programmes temps réel. Il faut donc utiliser un ramasse-miettes spécialisé (temps-réel), comme dans JamaicaVM [Siebert, 2007], une implémentation de machine virtuelle Java temps-réel. Cependant, la plateforme JamaicaVM n'est pas destinée pour systèmes embarqués avec des ressources très limités. Elle peut être déployée sur des hôtes équipés des systèmes d'exploitation tels que Linux, Solaris ou Windows. En effet, JamaicaVM est conçue pour les grosses machines.

En somme, les solutions existantes ne répondent pas aux contraintes inhérentes des systèmes embarqués avec des ressources très limitées. Afin de répondre aux contraintes mentionnées ci-dessus, nous proposons une solution de la mobilité qui se traduit par le développement d'une plateforme d'agents mobiles natifs pour systèmes embarqués homogènes. À notre connaissance, nous sommes les premiers à concevoir une plateforme d'agents à base des microcontrôleurs avec une mémoire aussi petite qu'un mégaoctet de RAM et de ROM.

4.2 Architecture de la plateforme d'agents mobiles pour systèmes embarqués

Dans cette section, nous exposons l'architecture de notre plateforme pour systèmes embarqués temps réel. Pour commencer, nous présentons notre choix de conception. Puis, nous décrivons la méthode et la directive de migration ainsi que le format de transfert d'agents mobiles.

4.2.1 Choix de conception

Pour déployer une application à base d'agents mobiles, il faut disposer d'une plateforme appropriée. Nous disposons de trois approches pour concevoir et implémenter une plateforme d'agents mobiles [Pierre, 2011]. La première consiste à utiliser un langage de programmation qui comporte des instructions pour les agents mobiles. La deuxième approche consiste à mettre en œuvre les agents mobiles comme des extensions du système d'exploitation. Enfin,

la dernière approche construit la plateforme comme une application spécialisée qui tourne au-dessus d'un système d'exploitation.

Nous avons choisi la deuxième approche. En effet, notre plateforme d'agents mobiles repose sur l'extension d'un noyau temps réel. Ce dernier est un logiciel permettant de coordonner l'exécution simultanée de plusieurs tâches en répartissant les ressources du système (processeur, mémoire, accès aux périphériques, etc.) et en fournissant des services (communication, synchronisation, etc.). Cependant, la machine ne dispose que d'un processeur unique qui exécute des instructions séquentiellement. Pour exécuter plusieurs tâches simultanément, on utilise le principe de l'entrelacement. On entrelace l'exécution des différentes tâches. Pour ce faire, il faut effectuer un ordonnancement des différentes tâches et changer l'attribution du processeur entre celles-ci. Ce concept précédemment décrit est communément appelé multitâche.

Cette approche contribue à la création d'applications modulaires. En effet, celle-ci permet surtout au programmeur de gérer la complexité en divisant des applications temps réel entre différentes tâches qui sont chacune responsable d'une partie du problème. Chaque tâche possède une priorité, un code à exécuter, une partie de la mémoire et une zone de pile. Grâce à cette approche, les applications temps réel sont plus faciles à concevoir et à maintenir. Comme illustrée à la figure 4.2, une tâche peut prendre une des cinq états suivants:

(1) Dormant: La tâche est toujours en mémoire mais n'est pas considérée par les opérations d'ordonnancement. Elle n'essaie donc pas de prendre le contrôle de l'UCT.
(2) Prêt: la tâche attend pour être assignée à l'UCT.
(3) Actif: La tâche est celle dont les instructions sont actuellement exécutées par l'UCT.
(4) En attente: La tâche attend un signal extérieur (la libération d'une ressource, l'expiration d'un délai, etc.) afin de poursuivre son exécution.
(5) Suspendu: la tâche possédait l'usage de l'UCT lorsqu'elle a été suspendue par une interruption. Autrement dit, l'exécution de la tâche a été suspendue par l'arrivée d'un signal d'interruption et le l'UCT est en train de traiter cette interruption.

Au niveau de la mise en œuvre, une tâche ressemble à n'importe quelle fonction en C/C++ à part quelques petites différences. Il existe deux types de tâches: une qui prend fin et l'autre qui s'exécute en boucle infinie. Dans la plupart des systèmes embarqués, les tâches prennent généralement la forme en boucle infinie. En outre, aucune tâche n'est autorisée à retourner comme les fonctions le permettent. Étant donné qu'une tâche est comme une fonction en C/C++, elle peut déclarer des variables locales.

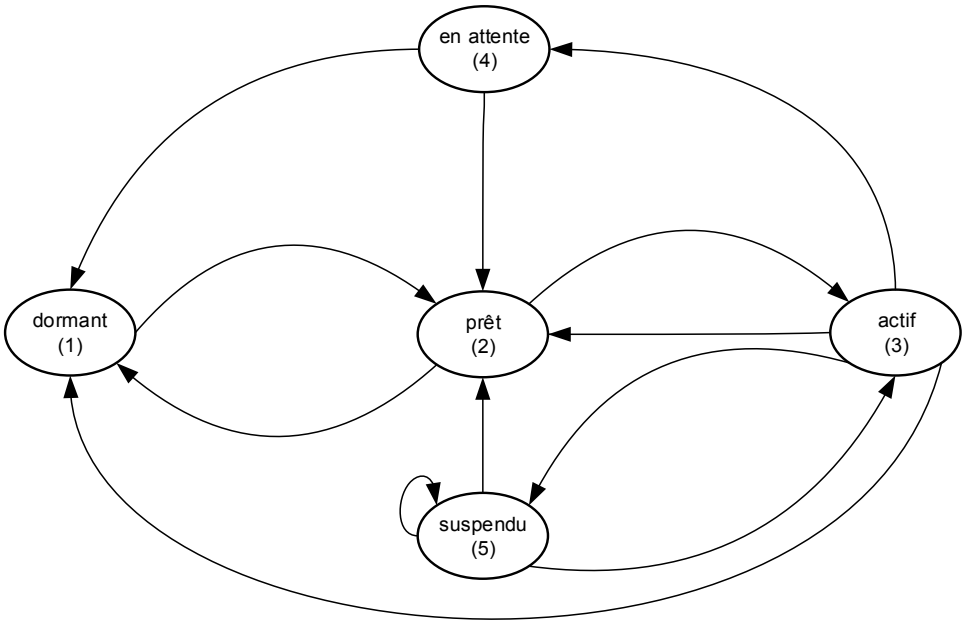


Figure 4.2 Cinq états de base d'une tâche

Les noyaux temps réel pour systèmes embarqués se distinguent par les caractéristiques suivantes:

- Ils ne nécessitent que peu de ressources.
- La description de la stratégie d'ordonnancement, du temps d'exécution maximum de chaque service et de l'ensemble des mécanismes internes capables d'affecter les temps de réponses (par exemple, le plus long intervalle de désactivation des interruptions par le noyau).
- L'utilisateur a la possibilité de programmer des opérations urgentes au sein de routines d'interruption.
- Le noyau fournit des services liés au temps: temporisateurs, exécution périodique de tâches.

- La configuration du noyau peut être paramétrée par le développeur.

4.2.2 Méthode de migration d'agents

Notre méthode de migration d'agents est basée sur l'extension des services de bas niveau d'un noyau temps réel. Les deux principales fonctions d'un noyau temps réel sont: l'ordonnancement et le changement de contexte d'exécution de tâches. L'ordonnanceur, aussi appelé le répartiteur, est la partie du noyau chargée de déterminer la tâche qui s'exécute. En effet, il attribue toujours le processeur à la tâche prête qui possède la plus haute priorité. Lorsqu'il est possible que plusieurs tâches de même priorité deviennent prêtes, plusieurs approches sont envisageables:

1. soit on alterne l'exécution de séquences bornées d'instructions de toutes ces tâches (une technique dite de «*time slicing*» en anglais, du fait que chaque tâche dispose de tranches de temps successives),
2. soit on attribue arbitrairement le processeur à une de ces tâches,
3. soit on évite cette situation en interdisant d'attribuer la même priorité à deux tâches distinctes.

Les deux premières stratégies compliquent la possibilité de déterminer précisément l'échéance d'une tâche. Par conséquent, nous privilégions dans ce projet la troisième approche. En effet, dans ce travail chaque tâche est assignée à une priorité en fonction de son importance.

Lorsqu'une tâche T_2 plus prioritaire que la tâche active T_1 passe de l'état suspendue à l'état prête, deux mécanismes sont possibles:

1. La tâche T_2 reste suspendue jusqu'à la complétion de T_1. L'ordonnanceur n'est alors pas préemptif. L'inconvénient est que le temps de réponse d'une tâche est affecté par le comportement des tâches moins prioritaires.
2. Le noyau bascule la tâche T_1 dans l'état suspendue et attribue le processeur à T_2. Dans ce cas l'ordonnanceur est dit préemptif.

Le noyau utilisé dans ce travail emploie un ordonnanceur préemptif. Ainsi, lorsqu'un événement indique qu'une tâche plus prioritaire est prête, le noyau sauvegarde le contexte de la tâche en cours (les registres du processeur) sur la pile afin de permettre une éventuelle reprise. Ensuite, le contexte de la tâche prioritaire est récupéré de son secteur de mémoire (sa pile) puis l'exécution reprend là où elle avait été interrompue. Comme nous montre la figure 4.3, chaque tâche possède sa propre pile. À tout moment, c'est à la tâche de plus haute priorité prête que l'ordonnanceur attribue au processeur. Avec ce type d'ordonnanceur, le temps de réponse des tâches est optimum et déterministe.

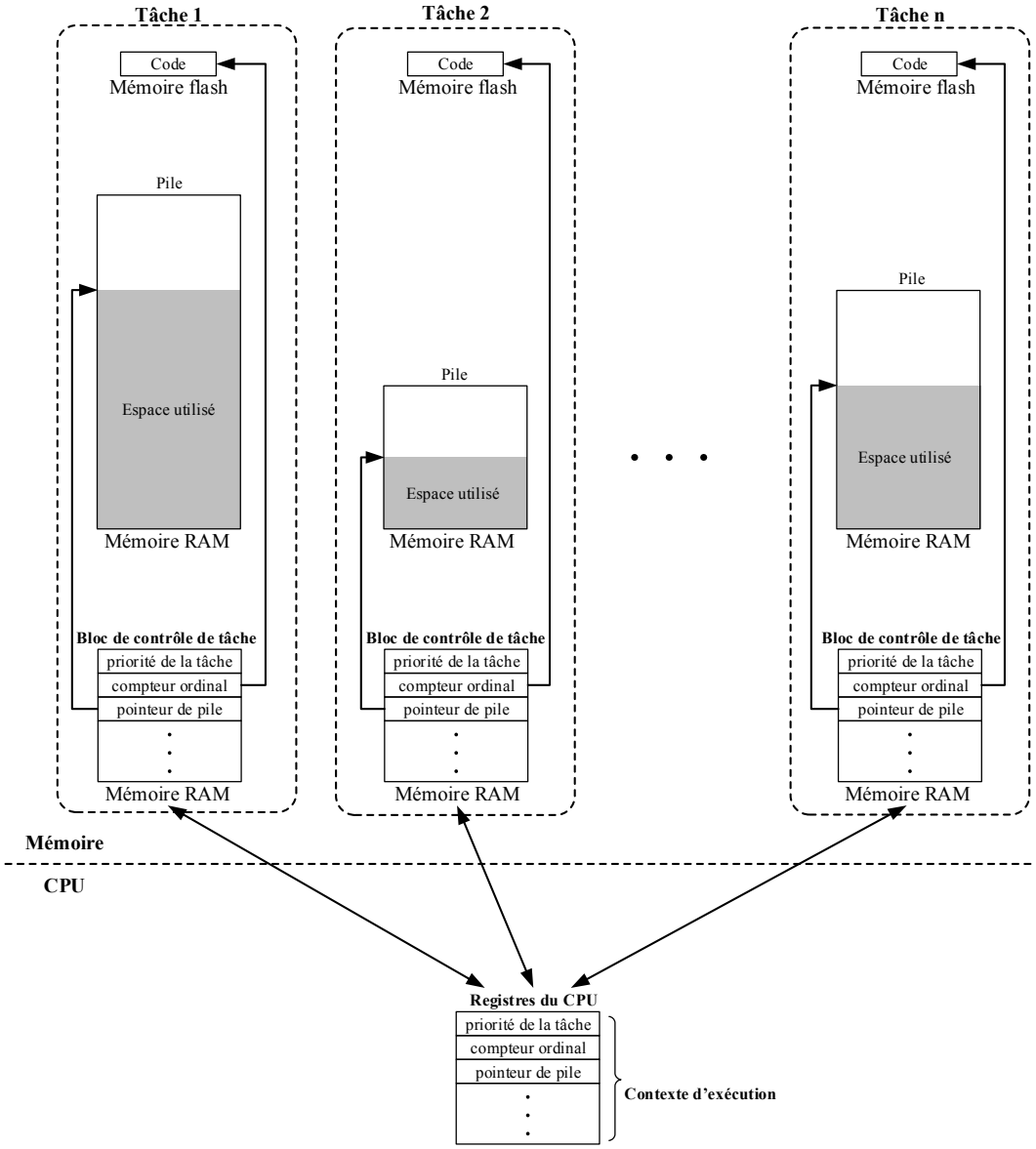


Figure 4.3 Environnement multitâche

Le changement de contexte peut être vu comme un service, parmi d'autres, qui ajoute un surcoût à l'application. Plus le processeur possède de registres, plus il y a du surcoût. Le temps requit par un changement de contexte est déterminé par le nombre de registres à sauver et à restaurer par l'UCT.

Comme précédemment mentionné, la plateforme μC/MAS repose sur l'extension d'un noyau temps réel en exploitant l'analogie qui existe entre le changement du contexte d'exécution de tâches par ce noyau et la mobilité d'agents. Le changement de contexte d'exécution est basé sur le mécanisme d'interruption. Celle-ci est un signal demandant au processeur de suspendre temporairement l'exécution de la tâche courante afin d'effectuer des opérations particulières. L'intérêt de ce mécanisme est qu'il permet d'implémenter une réaction à une sollicitation en respectant les exigences suivantes:

- offrir un délai de réponse très bref,
- une programmation indépendante du code en cours d'exécution.

L'ordonnanceur et le mécanisme d'interruption ont tous deux la capacité de transférer le contrôle d'un point à un autre du code. Il est important de ne pas mettre en conflit ces deux mécanismes. Pour ce faire, il faut suivre les deux règles suivantes:

1. Une routine d'interruption ne peut pas faire appel à un service qui a la possibilité de suspendre la tâche courante (par exemple, l'acquisition d'un sémaphore, la lecture depuis une queue d'attente, etc.);
2. Si une routine d'interruption appelle un service qui peut conduire à un changement de contexte (déclenché par l'ordonnanceur), alors le noyau doit être au courant que c'est une routine d'interruption qui effectue l'appel.

Le non-respect de cette règle peut conduire à suspendre l'exécution de routines d'interruption. Pour se conformer à ces règles, chaque routine d'interruption programmée par l'utilisateur doit appeler un service spécial au début et à la fin de son exécution, afin de signaler au noyau que les instructions exécutées appartiennent à une routine d'interruption et non à une tâche.

L'ordonnanceur est implémenté par une fonction appelée après une opération modifiant l'état des processus comme:

- la création, la destruction d'une tâche ou modification d'une priorité;
- l'acquisition ou libération d'un objet de communication;
- le passage d'une unité de temps.

L'interruption est un dispositif incorporé au séquenceur qui détecte les signaux. Ceux-ci arrivent de façon asynchrone. Cette interruption peut être déclenchée:

- soit par un composant extérieur au processeur:
  - une requête d'interruption signalée via une ou plusieurs broches dédiées,
  - un changement de valeur logique à une entrée désignée,
- soit par le processeur lui-même:
  - une échéance d'une temporisation,
  - une exception arithmétique ou un dépassement lors d'un calcul arithmétique,
  - une interruption logicielle, etc.

Les noyaux temps-réel offrent des services de temporisation, permettant à une tâche de suspendre son activité pendant un certain laps de temps. Effet, un composant spécial déclenche périodiquement une interruption particulière (horloge). Gérée par le noyau, celle-ci est plus prioritaire que les interruptions programmées par l'utilisateur et se présente comme les battements de cœur du système. Le délai de suspension d'une tâche est exprimé en nombre d'occurrences de cette interruption (battements, en anglais *ticks*).

Pour garder l'intégrité des données des différentes tâches, il est donc indispensable que le contexte d'exécution de chaque tâche soit préservé. Comme illustré à la figure 4.4, examinons le changement du contexte d'exécution d'une tâche T_1 de basse priorité au profit d'une tâche T_2 de plus haute priorité [Micrium, 2013] [Labrosse et al., 2011] [Labrosse, 2002]:

(1) La tâche T_1 de basse priorité s'exécute;

(2) Une interruption survient, le contexte de la tâche T_1 est sauvegardé sur la pile et la routine de service d'interruption (en anglais *Interrupt Service Routine, ISR*) appropriée est obtenue en chargeant le contenu d'un vecteur d'interruption;

(3) La routine de service d'interruption détermine la tâche prête la plus prioritaire qui va prendre le contrôle du processeur, en occurrence la tâche T_2;

(4) Lorsque la routine de service d'interruption termine son exécution, le noyau charge le contexte de la tâcheT_2 (c'est-à-dire la tâche la plus haute priorité) dans le processeur;

(5) La tâche T_2 s'exécute et effectue le traitement nécessaire;

(6) Lorsque la tâche T_2 termine son traitement, elle reboucle au début du code, puis se met en attente;

(7) La tâche T_1 reprend exactement au point où elle a été interrompue.

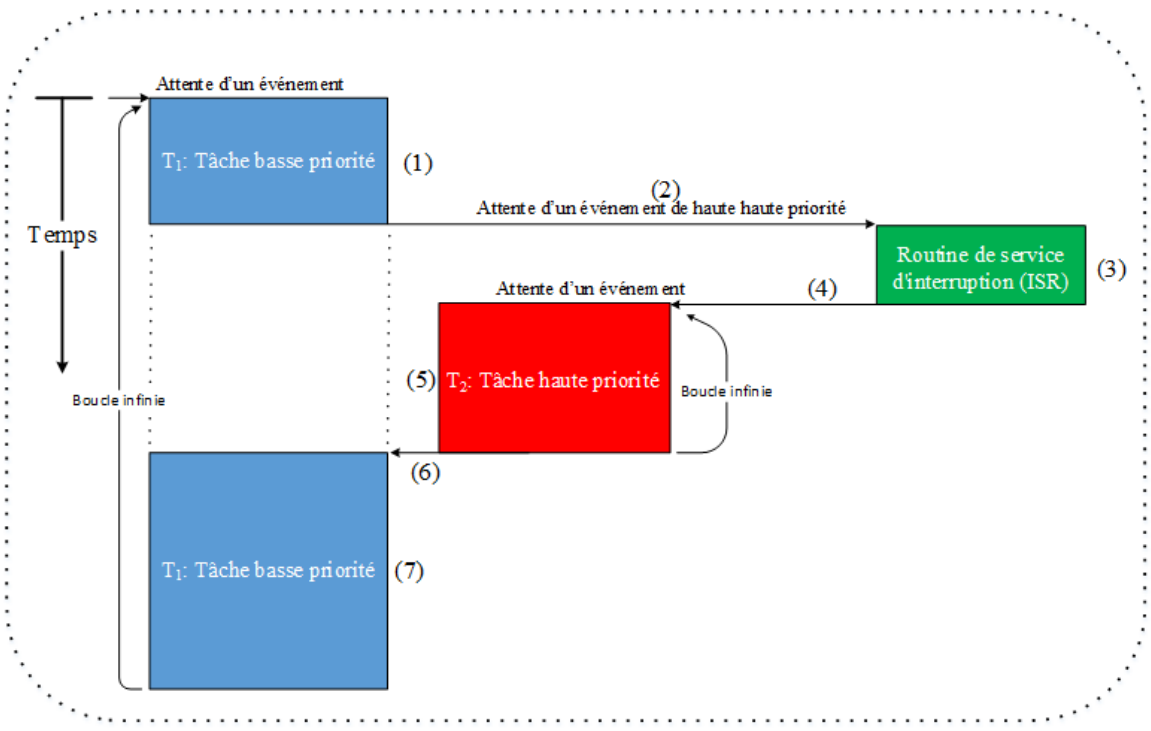


Figure 4.4 Changement du contexte d'exécution d'une tâche T_1 de basse priorité au profit d'une tâche T_2 de plus haute priorité

Lors de la migration d'une tâche *agent*, le noyau temps réel procède un changement de contexte. Pour provoquer un changement de contexte, la tâche se met en attente d'un événement. Pour ce faire, la tâche *agent* appelle une primitive comme *moveTo(adresse)*. Lorsque la tâche *agent* appelle la primitive *moveTo(adresse)*, le contenu de la variable *adresse* (qui est l'adresse de la machine de destination) est copié quelque part pour pouvoir l'utiliser au moment de la migration. Puis, la tâche *agent* se met en attente en utilisant des services de temporisation.

La figure 4.5 présente le code d'un agent mobile permettant d'effectuer dix fois le tour de trois nœuds dans un noyau temps réel, en l'occurrence µC/OS-II. Lors de la migration, l'agent mobile se déplace avec son contexte d'exécution incluant les adresses de trois nœuds à visiter. Dans chaque nœud, l'agent mobile affiche un message indiquant où il se trouve, puis il procède un certain nombre d'opération arithmétique et imprime les résultats sur le port série de l'ordinateur connecté au nœud avant d'aller à sa prochaine destination. Comme nous le montre ce code, l'agent mobile utilise les variables *AddrNode1, AddrNode2* et *AddrNode3* pour se rendre à sa destination. En fait, chacune de ces variables contient une adresse *MAC (Media Access Control)* d'un module sans fil (zigbee). La tâche *agent* débute son exécution sur le nœud *AddrNode1*.

Le noyau temps réel permet la gestion des communications entre les tâches. La façon la plus simple pour faire communiquer des tâches entre elles est par le biais des structures de données partagées. Les tâches peuvent ainsi utiliser des variables globales, des pointeurs, des listes chaînées, des tampons circulaires, etc. Le partage des données simplifie l'échange d'informations. Cependant, il faut s'assurer que chaque tâche dispose d'un accès exclusif aux données pour éviter les conflits et la corruption des données. Le mécanisme de sémaphore est la méthode la plus utilisée pour obtenir un accès exclusif aux ressources partagées. Cette méthode inventée par Edgser Dijkstra au milieu des années 1960 est un mécanisme supporté par la plupart des noyaux multitâches. Les sémaphores sont utilisés pour:

a) contrôler l'accès à une ressource partagée (exclusion mutuelle);
b) signaler l'occurrence d'un événement;
c) permettre à deux tâches à synchroniser leurs activités.

Un sémaphore est une clé que le code acquiert afin de poursuivre l'exécution. La tâche est suspendue si le sémaphore requis est utilisé (retenu) par une autre tâche. Il existe deux types de sémaphores: les sémaphores binaires et les sémaphores compteurs. Comme son nom l'indique, un sémaphore binaire ne peut prendre que deux valeurs: 0 ou 1. Un sémaphore compteur peut prendre des valeurs comprises entre 0 et 255 65 535 ou 4 294 967 295, selon que le type qui a mis en œuvre: 8, 16 ou 32 bits, respectivement. Avec la valeur du sémaphore, le noyau doit également garder la trace des tâches en attente de la disponibilité du

sémaphore. Il n'y a généralement que trois opérations qui peuvent être effectuées sur un sémaphore:

1) *initialize()* ou *create()*: Le sémaphore est initialisé à une valeur non nulle pour spécifier le nombre de ressources disponibles. À titre d'exemple, s'il y a 10 ressources, le sémaphore est initialisé à 10.
2) *wait()* ou *pend()*: Cette fonction attend un sémaphore.
3) *signal()* ou *post():* Cette fonction signale un sémaphore

```
static  void  TaskAgent (void *p_arg)
{
        INT8U err, MediumType = 1;
        INT32S i, j, k, temp, node, array_size = 10, a = 0x0123, b = 0x0567, c = 0x09AB, somme = 0x0DEF, round = 10;
        INT32S data[10] = {0x70, 0x20, 0x40, 0x80, 0x50, 0x45, 0x60, 0x90, 0x12, 0x10};
        INT8S string[100], message[100], messNode1[100], messNode2[100], messNode3[100], messNode4[100];
        INT8U AddrNode1[10] = {0x00, 0x13, 0xA2, 0x00, 0x40, 0x0A, 0x2F, 0xB4, 0xB9, 0xFA};
        INT8U AddrNode2[10] = {0x00, 0x13, 0xA2, 0x00, 0x40, 0x3E, 0x09, 0x55, 0xFF, 0xFE};
        INT8U AddrNode3[10] = {0x00, 0x13, 0xA2, 0x00, 0x40, 0x3E, 0x09, 0x59, 0x3B, 0x97};
        strcpy(message,"L'agent mobile a fini son itinéraire.\n\r");
        strcpy(messNode1,"L'agent mobile est maintenant dans le noeud 1.\n\r");
        strcpy(messNode2,"L'agent mobile est maintenant dans le nœud 2.\n\r");
        strcpy(messNode3,"L'agent mobile est maintenant dans le nœud 3.\n\r");
        somme += a + b + c + data[0];
        for(i = 0; i < round; i++){
                OSTaskMoveTo(AddrNode2, MediumType, &err);/* la primitive qui provoque la migration */
                if(err == OS_NO_ERR){
                        UART0WriteTxt(messNode2);/* affiche le message sur le port UART de l'ordinateur */
                        somme += i + a + b + c + data[i];
                        sprintf(string, "somme = %08X \n\r", somme);
                        UART0WriteTxt(string);/* affiche le message sur le port UART de l'ordinateur */
                }
                else UART0WriteTxt("Erreur de migration dans le noeud 2.\n\r");

                OSTaskMoveTo(AddrNode3, MediumType, &err); /* la primitive qui provoque la migration */
                if(err == OS_NO_ERR){
                UART0WriteTxt(messNode3);/* affiche le message sur le port UART de l'ordinateur */
                somme += i + a * b + c + data[i];;
                sprintf(string, "somme = %08X \n\r", somme);
                UART0WriteTxt(string);/* affiche le message sur le port UART de l'ordinateur */
                }
                else UART0WriteTxt("Erreur de migration dans le noeud 3.\n\r");

                OSTaskMoveTo(AddrNode1, MediumType, &err);/* la primitive qui provoque la migration */
                if(err == OS_NO_ERR){
                UART0WriteTxt(messNode1);/* affiche le message sur le port UART de l'ordinateur */
                somme += i + a + b * c + data[i];;
                sprintf(string, "somme = %08X \n\r", somme);
                UART0WriteTxt(string);/* affiche le message sur le port UART de l'ordinateur */
                }
                else UART0WriteTxt("Erreur de migration dans le noeud 1.\n\r");
        }
        UART0WriteTxt(message);
        while(1)
        {}
}
```

Figure 4.5 Code d'un agent mobile qui effectue dix fois le tour de trois nœuds

La valeur initiale du sémaphore doit être fournie lors de la création. La liste d'attente des tâches est toujours vide au départ. Une tâche qui désire le sémaphore effectue d'abord une opération d'attente. Si la valeur du sémaphore est supérieure à 0, la tâche l'obtient, par conséquent sa valeur est décrémentée et la tâche reprend son exécution. Si la valeur du sémaphore est 0, la tâche effectue une attente sur le sémaphore et, par conséquent, elle est suspendue. Cette tâche est placée dans une liste d'attente. La plupart des noyaux permettent de spécifier un délai d'attente. Si le sémaphore n'est pas disponible dans un certain laps de temps, la tâche qui le demandait reprend son exécution et un code d'erreur (indiquant qu'un délai d'expiration, en anglais *timeout,* s'est produit) est renvoyé à l'appelant.

Une tâche relâche (libère) un sémaphore en effectuant une opération *signal()*. Si aucune tâche n'attend le sémaphore, sa valeur est simplement incrémentée. En revanche, si une tâche est en attente du sémaphore, elle est passée dans l'état prêt et la valeur du sémaphore n'est pas incrémentée. Selon le noyau, la tâche qui recevra le sémaphore est:

a) soi la tâche la plus haute priorité en attente du sémaphore,
b) soi la première tâche qui a demandé le sémaphore.

Certains noyaux permettent de choisir l'une ou l'autre des méthodes lors de l'initialisation du sémaphore. Le noyau µC/OS-II supporte seulement la première méthode. Si la tâche prête a une priorité plus élevée que celle qui est en cours, un changement de contexte se produit. Dans le noyau µC/OS-II, l'ordonnanceur attribue toujours au processeur la tâche prête de plus haute priorité.

Un noyau ajoute des surcoûts à un système parce que les services fournis par le noyau ont besoin de temps pour s'exécuter. Les surcoûts dépendent du nombre de fois que les services sont appelés. Dans une application bien conçue, un noyau utilise entre 2 % et 5 % du temps de l'UCT [Labrosse et al., 2011] [Labrosse, 2002]. La migration de la tâche *agent* n'ajoute aucun surcoût sur les performances lors de l'exécution. Dans notre système, la capture est procédée lors de la migration de la tâche *agent* et par conséquent aucun traitement ne se fait lors de l'exécution normale.

Dans notre système, nous avons appliqué la mobilité de l'agent au niveau noyau pour avoir un accès direct et rapide à l'information à transférer. Ceci implique des extensions du noyau temps réel. Nous avons étendu le noyau temps réel en ajoutant des mécanismes permettant de:

- Capturer l'état courant:
  - Décider un changement de contexte d'exécution du processeur: ceci est initié par le code de *tâche agent*;
  - Extraire le contexte d'exécution de *tâche agent*;
- Transférer le contexte d'exécution de *tâche agent*;
- Restaurer le contexte d'exécution de tâche *agent*: La création de *tâche agent* avec un état initial pour que l'exécution reprenne le point où elle a été arrêtée.

Les mécanismes de capture, de transfert et de restauration de tâche en cours d'exécution sont intégrés aux fonctionnalités du noyau temps réel permettant ainsi la mobilité d'agents logiciels au sein d'une grappe de systèmes embarqués. La figure 4.6 présente la structure de la plateforme d'agents mobiles µC/MAS.

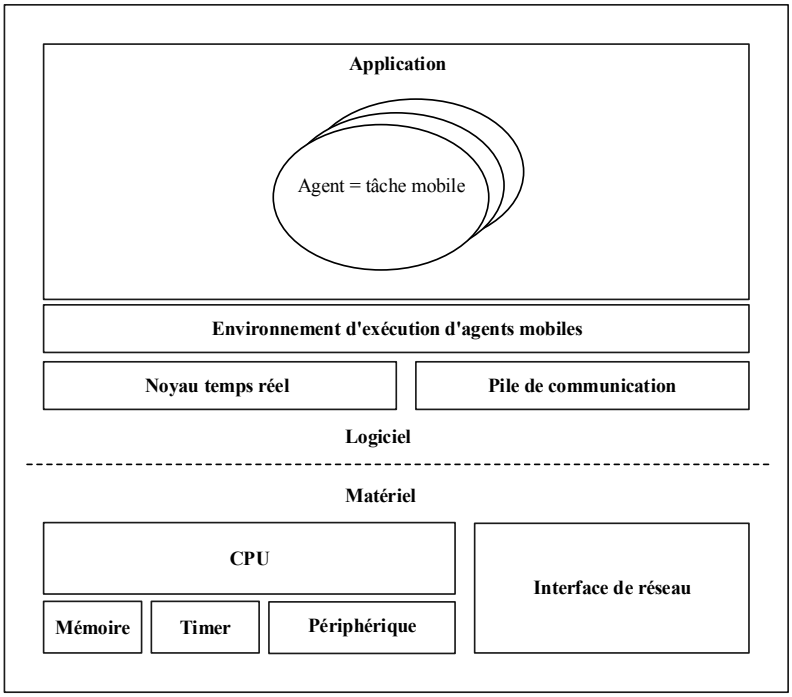


Figure 4.6 Structure de la plateforme d'agents mobiles µC/MAS

Les agents de la plateforme µC/MAS utilisent un code natif (le langage C et l'assembleur) et peuvent se déplacer aussi bien dans un réseau filaire que dans un réseau sans fil. Rappelons que le code natif s'appuie directement sur le processeur sous-jacent et, par conséquent, s'exécute beaucoup plus rapidement qu'un code interprété. Ceci est un point important dans le cas des systèmes embarqués avec des ressources très limités. Nos agents sont réalisés sur une plateforme pour systèmes embarqués homogènes. Ainsi, un agent mobile ne peut se déplacer que vers un microcontrôleur de même architecture physique ainsi qu'un même environnement d'exécution.

Dans le contexte de la plateforme µC/MAS, un agent est une tâche qui est capable de migrer d'un nœud à un autre. Lorsqu'un agent décide de migrer, il suspend son exécution sur le nœud courant, le nœud source. Puis, si le code de l'agent n'est pas déjà présent sur le site cible, il sera chargé à partir d'un serveur. Ensuite, les données représentant l'état de l'agent sont transférées du nœud source vers le nœud de destination. Une fois que l'agent arrive au nœud de destination, il reprend son exécution là où il avait été interrompu sur le nœud source. La figure 4.7 présente la migration des différents composants d'un agent mobile.

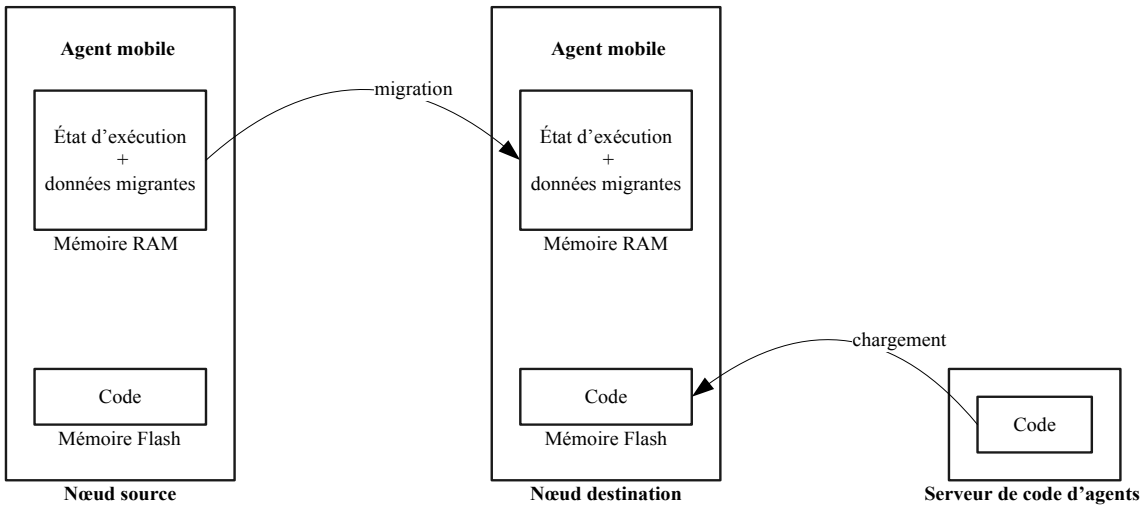


Figure 4.7 Migration des différents composants d'un agent mobile

La migration du contexte d'exécution de la tâche *agent* d'un nœud source vers un nœud destination suit les étapes suivantes:

(1) La tâche *agent (*T_1*)* qui s'exécute sur le nœud source décide de migrer. Elle appelle la primitive *OSTaskMoveTo(AddrNode2, MediumType, &err);*. En fait, cette primitive procède deux actions: Elle copie le contenu AddrNode2 dans une variable globale et se met en attente.

(2) Cet état d'attente provoque une interruption, le contexte de la tâche T_1 est sauvegardé sur la pile. La routine de service d'interruption appropriée est obtenue en chargeant le contenu d'un vecteur d'interruption.

(3) La routine de service d'interruption détermine la tâche prête la plus prioritaire qui va prendre le contrôle du processeur, en occurrence la tâche T_2 (tâche serveur);

(4) Lorsque la routine de service d'interruption termine son exécution, le noyau charge le contexte de la tâche T_2 dans le processeur.

(5) T_2 prend une priorité plus élevée que T_1. Puis elle capture le contexte d'exécution de T_1 (c'est-à-dire la pile et le bloc de contrôle de tâche, TCB).

(6) Elle transfère le contexte d'exécution de T_1 vers le nœud de destination.

(7) La tâche *serveur* (T_2) du nœud destination reçoit le contexte d'exécution. Elle crée une nouvelle tâche *agent* (T_1) avec le contexte d'exécution reçu.

(8) La tâche *serveur* du nœud destination envoie un message à T_2 du nœud source. Si le message indique que l'opération du transfert et la création de la tâche *agent* sont en succès, la tâche *serveur* du nœud source supprime la tâche *agent* et les données reliées, sinon on revient à l'étape (6).

(9) La tâche *serveur* du nœud destination prend une priorité plus basse que T_1. Ce changement de priorité provoque une interruption. Le noyau sauvegarde le contexte d'exécution de T_2 et passe à la routine de service d'interruption.

(10) La routine de service d'interruption détermine la tâche prête la plus prioritaire qui va prendre le contrôle du processeur, en occurrence la tâche T_1;

(11) Le noyau restaure le contexte de la tâche la plus prioritaire dans le processeur, c'est-à-dire le contexte de la tâche *agent* (T_1).

(12) La tâche *agent* du nœud destination reprend exactement au point où elle a été interrompue.

La figure 4.8 présente la Migration de la tâche agent du nœud source vers le nœud destination.

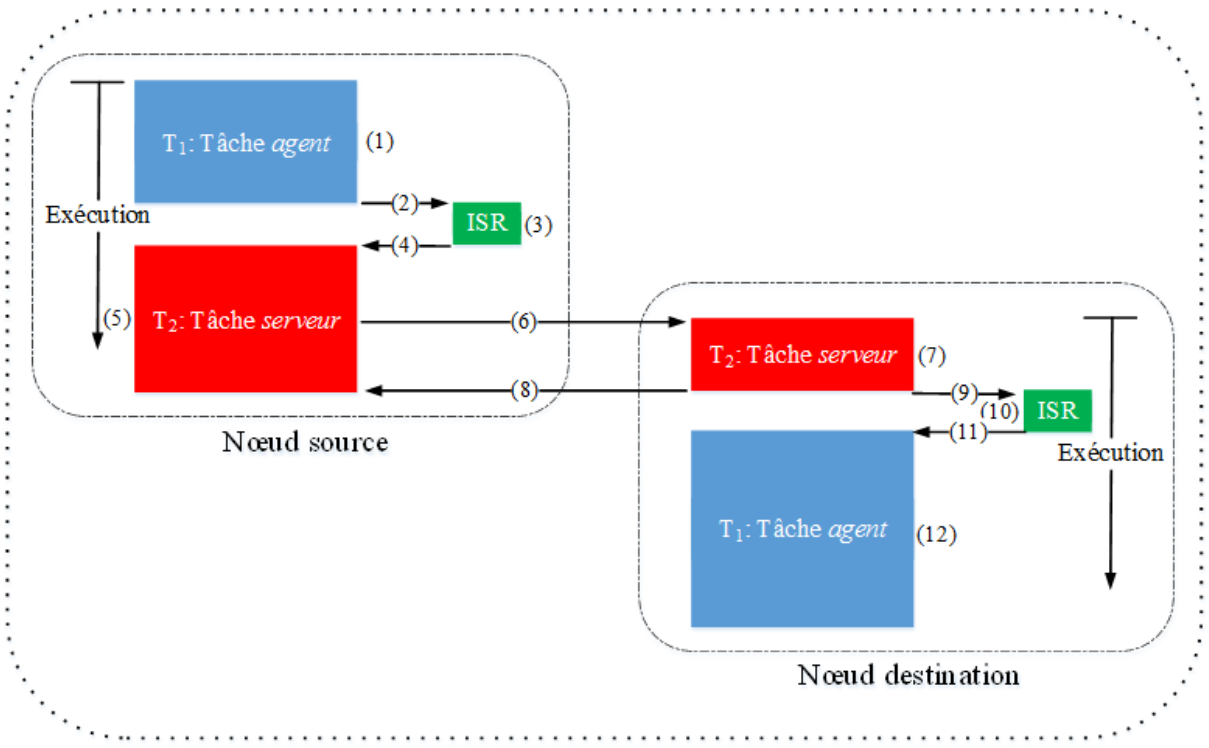


Figure 4.8 Migration de la tâche agent du nœud source vers le nœud destination

4.2.3 Directive de migration d'agents

La directive présente une démarche à utiliser pour la préparation des données représentant l'agent en vue de leur migration. Dans son cycle de vie, une tâche *agent* utilise des variables locales et/ou globales. La question qu'on doit se poser maintenant est: faut-il migrer avec la tâche *agent* toutes les variables qu'elle a utilisées dans le nœud source? La réponse à cette question est non car il faut utiliser les ressources de façon optimale. En effet, le but de la directive de migration est d'optimiser les ressources telles que l'espace mémoire utilisé, le temps de transfert entre les nœuds, etc. En suivant cette directive, un développeur d'application peut choisir les données essentielles qui migrent avec l'agent en les plaçant dans les segments de mémoire appropriés.

Dans le contexte d'une mobilité forte d'une tâche *agent*, les éléments à migrer sont le *code*, les *données courantes* et l'*état d'exécution (la pile et le bloc de contrôle de tâche)*. Comme nous le présente la figure 4.9, une tâche *agent* est constitué d'un segment de:

- pile (en anglais *stack*) permettant de stocker les appels de fonctions avec leurs paramètres et leurs variables locales. Lors d'un retour de fonction, les paramètres et variables sont dépilés.
- tas (en anglais *heap*) réservé aux allocations dynamiques de mémoire.
- variables globales et statiquement non-initialisées sont par défaut mises à zéro. Cette section est appelée en anglais BSS (pour *block started by symbol*).
- variables globales et statiques initialisées (en anglais *data*).
- texte (code) qui correspondant aux instructions du programme à exécuter. Ce code est le fichier exécutable produit par le compilateur ou l'assembleur.

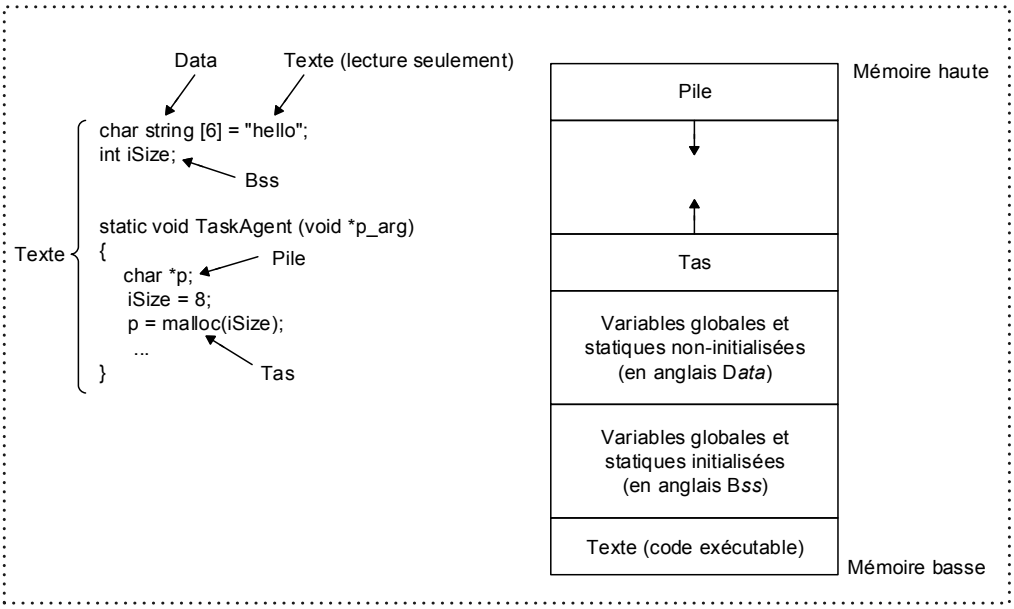


Figure 4.9 Structure générale d'une tâche agent en mémoire

À part la pile, les segments (les tas, les données non-initialisées et les données initialisées) sont partagés entre les différentes unités de l'application (les tâches). En effet, seulement le segment de la pile est propre à chaque tâche. Dans notre approche de migration, le contexte d'exécution migre automatiquement avec la tâche *agent*. En revanche, le concepteur de l'application doit décider les données (variables) globales qui doivent migrer avec la *tâche agent*. Comme ces données sont dispersées dans l'espace de la mémoire, nous définissons une directive de migration. Pour ce faire, nous exploitons les classes de stockage qui existent déjà dans le langage C/C++ de façon totalement nouvelle en spécifiant des blocs des données

migrantes. En effet, ces blocs permettent d'assembler (ou grouper) les données dispersées de la tâche *agent* dans l'espace de la mémoire. Ainsi, il est plus facile de migrer des données groupées dans un bloc plutôt que des données dispersées dans l'espace de la mémoire du système.

Dans son cycle de vie, un agent utilise différents types des données. Pour gérer la migration, nous classons les données en différentes catégories en se basant sur la classification des variables existantes en C/C++. Cette classification peut être appliquée dans d'autres langages. En C/C++, les variables sont classées en différentes catégories selon la manière dont elles sont créées et la manière dont elles pourront être utilisées. Les différents aspects que peuvent prendre les variables constituent ce que l'on appelle leur classe de stockage. La classe de stockage d'une variable permet de spécifier sa durée de vie et sa place en mémoire. Une description détaillée de la classification des variables existantes en C/C++ se trouve dans l'annexe B de ce document.

Dans le but de définir et donner un cadre de migration des données avec l'agent, nous commençons par classer les variables en locales et globales. Les variables locales sont créées à l'intérieur d'un bloc d'instructions, dans le cas de notre système dans la tâche *agent*. Les variables globales sont en revanche déclarées en dehors de tout bloc d'instructions dans la zone de déclaration globale du programme. Les variables locales et globales ont différentes durées de vie et différentes portées selon leurs emplacements en mémoire. La portée d'une variable est la zone du programme dans laquelle elle est accessible. La portée des variables globales est tout le programme tandis que la portée des variables locales est le bloc d'instructions dans lequel elles ont été créées.

En deuxième lieu, nous classons les variables locales en *automatiques* et *dynamiques*. La zone contenant les variables *automatiques* est gérée par la pile. Les variables *dynamiques* sont stockées dans des partitions de blocs de mémoire. La notion des partitions de blocs de mémoire est décrite un peu plus bas.

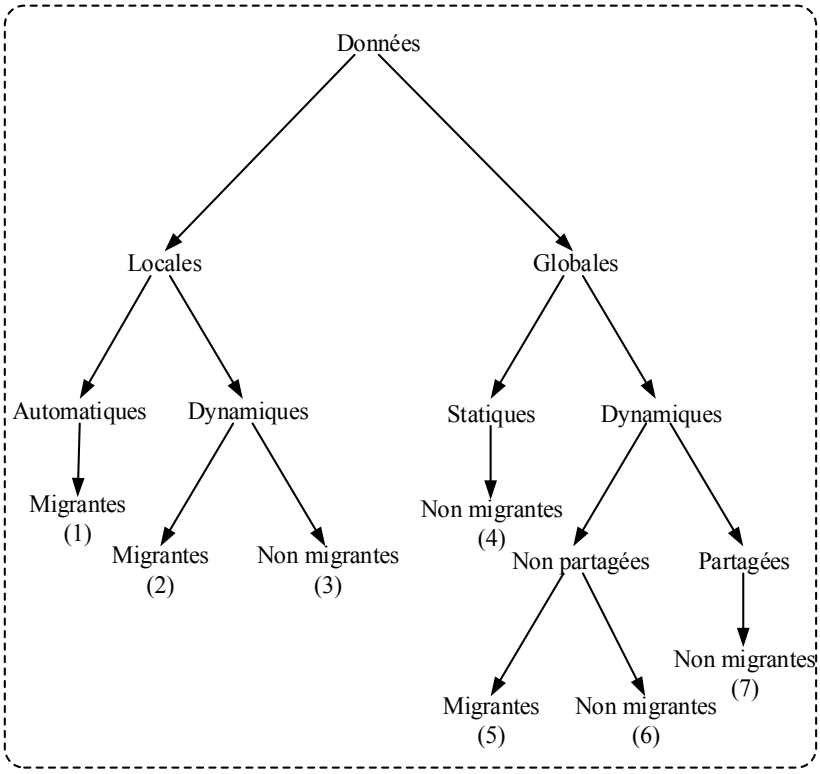


Figure 4.10 Classification des données interagissant avec l'agent

La figure 4.10 présente une classification des données à l'aide d'un arbre binaire où chaque feuille spécifie la migration ou non de ces données avec l'agent:

1. Les *variables locales automatiques* représentées par la *feuille (1)* migrent automatiquement avec l'agent puisqu'elles se trouvent sur la pile.
2. Le développeur de l'application décide la migration ou non des *variables locales dynamiques* représentées par les *feuilles (2) et (3)*. Notons que ces variables se trouvent sur la pile mais les adresses pointées sont dans le *segment de tas*. Sur le nœud source, les blocs de mémoire contenant ces variables doivent être remis à leur partition une fois que la migration est effectuée. Pour ce faire, il faut utiliser les fonctions que le noyau nous offre pour libérer les blocs.
3. Les *variables globales statiques* représentées par la *feuille (4)* ne migrent pas avec l'agent. En effet, si le développeur désire utiliser des *variables globales,* il peut obtenir via des blocs de mémoire pour ensuite libérer après la migration de l'agent. Le terme statique employé ici fait opposition au terme dynamique. Il ne s'agit pas du mot clé «*static*» qu'on utilise en C.

4. Le développeur de l'application décide la migration ou non des *variables globales dynamiques non partagées* représentées par les feuilles *(5) et (6)* avec l'agent. Les blocs de mémoire contenant ces variables doivent être remis à leur partition (libération) après utilisation. Les *variables globales dynamiques non partagées* sont celles exclusivement utilisées par l'agent.
5. Les variables *globales dynamiques partagées* représentées par les feuilles *(7)* ne migrent pas avec l'agent. Les *variables globales dynamiques partagées* sont celles non seulement utilisées par l'agent mais également par d'autres unités de l'application.

Le but de l'arbre binaire est de proposer une directive de migration en se basant sur la classification des variables existantes en C/C++. Bien qu'un concepteur d'applications à base d'agents mobiles puisse employer cette directive de migration, nous suggérons de réduire au strict minimum l'utilisation des variables globales. Nous encourageons l'utilisation des variables locales qui migre automatiquement avec la tâche *agent*. En effet, pour migrer les variable globales, il faut utiliser la directive de migration qui permet de les rassembler et de les transférer comme vous allons décrire plus bas.

Le mécanisme de la partition de mémoire proposé par certains noyaux temps réel tels que µC/OS-II et µC/OS-III permet d'éviter l'utilisation des fonctions *malloc()* et *free()* [Labrosse, 2011] [Labrosse, 2009] [Labrosse, 2002]. L'utilisation de ces fonctions dans un système embarqué temps réel est dangereuse puisqu'il n'est pas toujours possible d'obtenir une zone de mémoire contiguë en raison de la fragmentation externe inhérente. Pour pallier à cette problématique, nous utilisons des mécanismes permettant d'offrir des blocs de mémoire de taille fixe dans une zone de mémoire contiguë. Dans une partition, tous les blocs de mémoire ont la même taille comme illustré à la figure 4.11.

Le mécanisme d'allocation et de libération de ces blocs de mémoire est déterministe et se fait à un temps constant. La partition est généralement allouée de manière statique (comme un tableau), mais peut être également puisée dans le bassin dynamique conventionnel, c'est-à-dire dans le *tas*, sans être libérée (sans jamais utiliser la fonction *free()*).

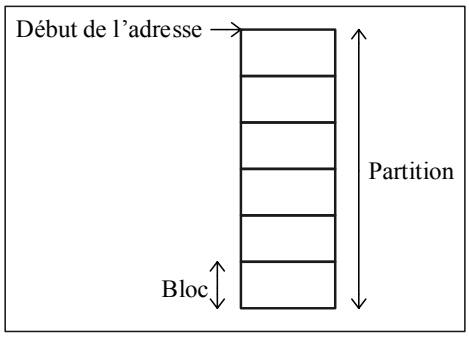


Figure 4.11 Partition de blocs de mémoire à taille fixe

Dans une application, il peut exister de multiples partitions de mémoire. Cependant, chaque bloc de mémoire spécifique doit toujours être retourné à la partition d'où il provient. En effet, il faut utiliser les fonctions que le noyau temps réel (µC/OS-II ou µC/OS-III par exemple) nous offre pour allouer et libérer un bloc. Ce type de gestion de la mémoire n'est pas assujetti à la fragmentation. Avant d'utiliser une partition de blocs de mémoire, il faut d'abord la créer. Ceci permet au noyau de saisir la partition de blocs de mémoire afin de pouvoir gérer leur allocation et libération. Dans le noyau temps réel comme µC/OS-II [Labrosse, 2002], l'appel de la fonction *OSMemCreate()* crée la partition de mémoire telle qu'illustrée à la figure 4.12.

```
OS_EVENT        *SemaphorePtr1;
OS_MEM          *PartitionPtr1;
INT8U           Partition1[50][128];

OS_EVENT        *SemaphorePtr2;
OS_MEM          *PartitionPtr2;
INT32U          Partition2[40][64];

INT8U *Bloc1;
INT32U *Bloc2;

int main (void)
{
 INT8U  err;
 OSInit();  /* Initialisation du noyau temps réel µC/OS-II. */

SemaphorePtr1 = OSSemCreate(50);  /*Création d'un sémaphore avec un compte initial correspondant au nombre de blocs dans la partition*/
PartitionPtr1 = OSMemCreate(Partition1, 50, 128, &err); /*Création d'une partition de 50 blocs de 128 octets, donc un total = 6400 octets*/

SemaphorePtr2 = OSSemCreate(40); /*Création d'un sémaphore avec un compte initial correspondant au nombre de blocs dans la partition*/
PartitionPtr2 = OSMemCreate(Partition2, 40, 64, &err);/*Création d'une partition de 40 blocs de 4*64 octets, donc un total = 10240 octets*/
                                        •
                                        •
                                        •
  OSStart();      /* Le début du multitâche (c'est-à-dire donner le contrôle au noyau, µC/OS-II dans cette exemple) */
}
```

Figure 4.12 Exemple d'un code permettant la création de deux différentes partitions

```
static  void  TaskAgent (void *p_arg)
{
        INT8U err;
        INT32U i;
        INT8S string[200];

        OSSemPend(SemaphorePtr1, 0, &err); /*Attendre le sémaphore SemaphorePtr1*/
        Bloc1 = OSMemGet(PartitionPtr1, &err); /*L'acquisition du bloc Bloc1*/

        for(i = 0; i <128; i++) Bloc1 [i] = '\0'; /*Initialisation*/
        strcpy(Bloc1,"123456789ABCDEF123456789ABCDEFDEF\n\r"); /*Mettre les données dans Bloc1*/

        OSSemPend(SemaphorePtr2, 0, &err);/*Attendre le sémaphore SemaphorePtr2*/
        Bloc2 = OSMemGet(PartitionPtr2, &err); /*L'acquisition du bloc Bloc2*/
        for(i = 0; i <64; i++)  Bloc2 [i] = 0;  /*Initialisation du bloc Bloc2*/
        for(i = 0; i <64; i++) Bloc2 [i] += i; /*Mettre des données dans le bloc*/

        UART0WriteTxt("\n\r");
        UART0WriteTxt(Bloc1);/*Afficher le contenu de chaque bloc dans l'UART 0*/

        UART0WriteTxt("\n\r");
        for(i = 0; i <64; i++) {
                sprintf(string, "%04X ", Bloc2[i]); /*Afficher le contenu du bloc dans l'UART 0*/
                UART0WriteTxt(string);
        }
        OSMemPut(PartitionPtr1, Bloc1);/* Retourner le bloc Bloc1 à sa partition*/
        OSSemPost(SemaphorePtr1);/*Libérer le sémaphore SemaphorePtr1*/
        OSMemPut(PartitionPtr2, Bloc2);/* Retourner le bloc Bloc2 à sa partition*/
        OSSemPost(SemaphorePtr2); /*Libérer le sémaphore SemaphorePtr2*/

          •
          •
          •

        while(1)
        {}
}
```

Figure 4.13 Exemple d'un code permettant l'acquisition et la libération des blocs de partition

Le code de l'application peut demander un bloc de mémoire d'une partition en appelant la fonction *OSMemGet()* du noyau temps réel, en l'occurrence µC/OS-II, comme nous le présente la figure 4.13. Le code suppose que la partition est déjà créée. Lorsque la tâche *agent* s'exécute, elle obtient un bloc de mémoire que si un sémaphore est disponible. En effet, le bloc de mémoire peut être obtenu si et seulement si le sémaphore est disponible. Il n'est pas nécessaire de vérifier le code d'erreur de la fonction OSSemPend() parce que le noyau peut donner le contrôle du processeur à cette tâche si et seulement si un bloc de mémoire est libérée

(puisque le délai d'attente est 0). En outre, il n'est pas nécessaire de vérifier le code d'erreur de la fonction *OSMemGet()* pour la même raison. On doit avoir au moins un bloc dans la partition pour que la tâche *agent* reprenne son exécution.

Le mécanisme de sémaphore fonctionne sommairement comme suit. À la création du sémaphore, on fournit le nombre de blocs dans la partition à l'aide de primitive *OSSemCreate()*. Pour prendre un bloc, la primitive *OSSemPend()* est appelée. Lorsqu'il n'y a pas de blocs dans la partition, la tâche, en occurrence *TaskAgent*, se met en attente. Celle-ci permet la suspension de la tâche. Lorsqu'une tâche appelle la primitive *OSSemPost()*, elle indique qu'elle a mis un bloc dans la partition. Celui-ci peut être pris par une tâche qui était suspendue en attente de blocs. L'appel de la primitive *OSSemPost()* permet donc de débloquer une tâche.

Notons que la primitive *OSMemPut()* est appelée pour remettre le bloc dans sa partition avant de libérer le sémaphore. Nous appelons les *blocs des données migrantes* les blocs contenant des *variables dynamiques* des tâches *agent*. La figure 4.14 (a) illustre comment les données pourraient être dispersées dans la mémoire sans l'application de notre directive. En revanche, avec l'application de notre directive de migration (voir la figure 4.14 (b)), les données de la tâche *agent* sont mises dans une partition de blocs de données. Ceci facilite l'accès et ainsi le transfert des données de manière efficace vers la destination.

Lorsqu'une tâche se déplace d'un nœud à un autre, il n'est pas nécessaire de migrer la partition au complet. Il faut seulement migrer les blocs des données utilisées et libérer les restes. Par exemple, chaque bloc est peut être vu comme un tableau de 64 caractères dans l'architecture de la machine ARM, par exemple. L'architecture ARM est fortement inspirée des principes de conception RISC.

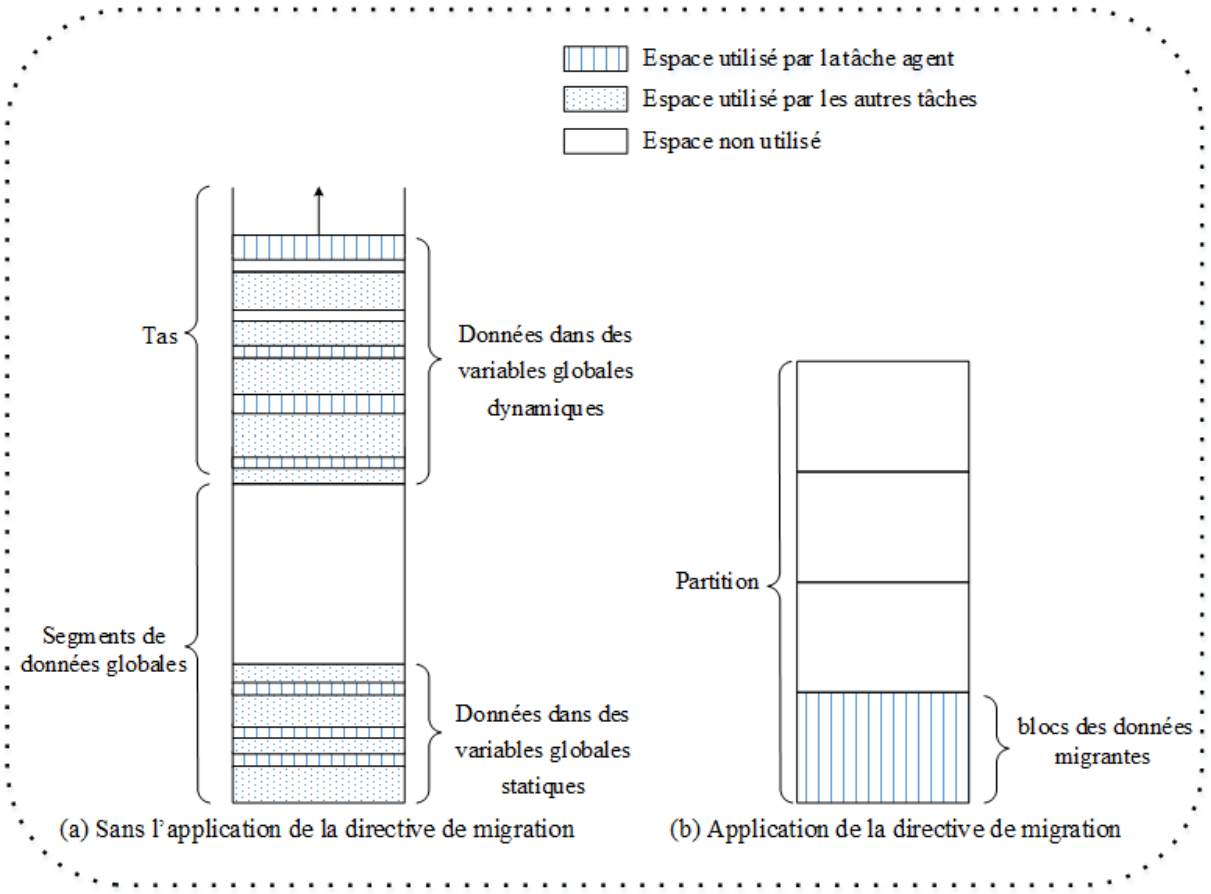


Figure 4.14 Application de la directive de migration ou non

Lors de migration d'une tâche *agent*, l'espace de mémoire (la position) utilisé par la pile du nœud source n'est pas toujours disponible à la destination. Il faut donc relocaliser (placer dans une adresse différente de celle du nœud source) la pile de chaque tâche au fur et à mesure qu'elle se présente à la destination. Pour les même raisons que mentionnées précédemment, nous utilisons encore une fois le mécanisme des partitions de blocs afin d'allouer un espace de mémoire pour la pile. La figure 4.15 présente la relocalisation d'une pile de tâche d'un nœud à un autre.

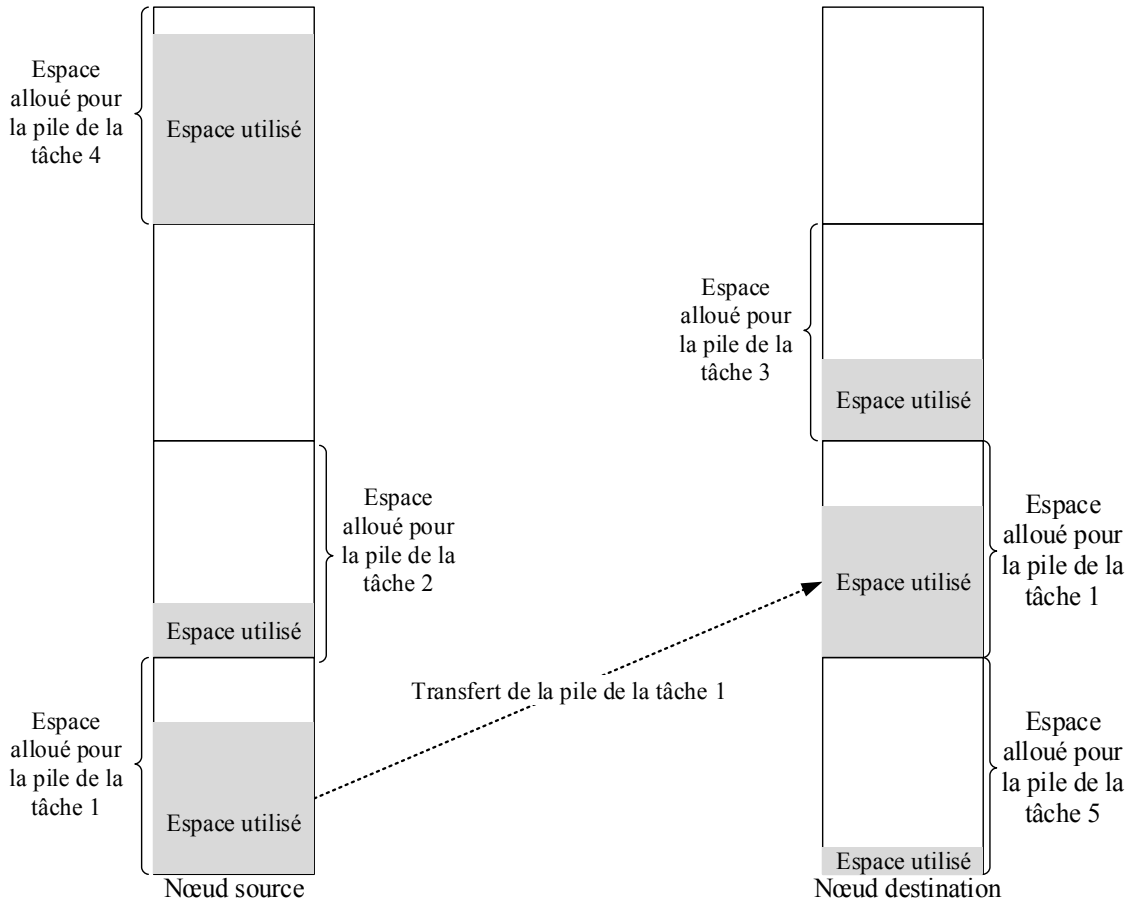


Figure 4.15 Relocalisation d'une pile de tâche d'un nœud à un autre

Une partition des blocs des données peut être créée de manière dynamique (avec *malloc()*) ou statique. Dans µC/MAS, nous optons pour les allocations statiques. Ceci consiste à fixer le nombre et la taille des partitions de blocs de mémoire. Cette approche présente des avantages dans le contexte des systèmes embarqués. En effet, l'utilisation des blocs de mémoire fixe permet l'allocation et la libération de ceci d'une façon unitaire en évitant ainsi la fragmentation de mémoire. Le nombre de blocs de mémoire est directement dépendant du nombre d'agents présents sur le nœud à un moment donné. Le besoin en mémoire peut ainsi être déterminé à l'avance en fonction de la structure et du nombre des agents en circulation dans le réseau. Ainsi, il devient possible de produire une plateforme d'agents mobiles fiables pour systèmes embarqués dans un contexte d'environnement homogène.

4.2.4 Format de transfert d'agents

Lors de migration, l'agent doit disposer d'un format de transfert. Il existe un certain nombre de formats de transfert de programmes entres les machines. Nous pouvons citer les formats *Intel HEX* [Intel Corporation, 1988] et *S-Record* de Motorola [Bernett et al., 1987], par exemple. Nous avons opté le format *XML* en combinaison avec celui *Intel HEX*. Nous avons choisi le format *Intel HEX* parce qu'il est le plus utilisé. En outre, notre compilateur génère ce format.

Comme, un agent mobile est constitué de trois composants, à savoir un code, un contexte d'exécution (la pile et TCB) et des données courantes, nous avons eu besoin un format permettant une intégration. XML est un format ouvert qui permet une intégration élégante de différents composants de l'agent mobile. Avant de présenter le format *Intel HEX*, nous allons d'abord décrire celui de *S-Record* de Motorola. Ce dernier est un format de représentation de fichier binaire en ASCII. À l'origine, le format *S-Record* de Motorola était utilisé pour la programmation du microprocesseur Motorola 6800. Il est utilisé depuis en systèmes embarquée comme son concurrent, le format Intel HEX. En effet, les deux formats sont textuels et offrent de nombreux avantages sur le format binaire: ils peuvent être imprimés, inspectés ou modifiés avec un éditeur de texte ordinaire. Ils sont utilisés pour le transfert de codes vers les programmateurs d'EPROM en communication série (RS-232).

Format S-Record

Un fichier en format S-Record est constitué de chaînes de caractères hexadécimaux, appelées aussi enregistrements. Tous les nombres hexadécimaux sont en *big endian*. Comme nous le présente le tableau 4.1, un enregistrement en format S-Record suit la structure suivante:

- *Type* de la chaine;
- *Longueur* de la chaine;
- *Adresse* mémoire où sont les données;
- *Données*;
- *Somme de contrôle* (en anglais, checksum).

Tableau 4.1 Caractéristiques des champs d'un enregistrement (ou une chaîne)

Enregistrement	Nombre de caractères	Contenu
type	2	• S0: L'enregistrement d'en tête. Le champ de données peut contenir une description. L'adresse est normalement nulle. • S1: L'enregistrement de données a une adresse sur 2 octets. • S2: L'enregistrement de données a une adresse sur 3 octets. • S3: L'enregistrement de données a une adresse sur 4 octets. • S5: L'enregistrement qui indique sur le nombre d'enregistrements S1, S2 et S3 transmis dans un bloc en particulier. Ce nombre apparait dans le champ de l'adresse. Il n'y a aucune donnée. • S7: L'enregistrement final d'un bloc d'enregistrements S3. L'adresse peut optionnellement contenir une adresse sur 4 octets qui pointe sur une instruction. Il n'y a aucune donnée. • S8: L'enregistrement final d'un bloc d'enregistrements S2. L'adresse peut optionnellement contenir une adresse sur 3 octets qui pointe sur une instruction. Il n'y a aucune donnée. • S9: L'enregistrement final d'un bloc d'enregistrements S1. L'adresse peut optionnellement contenir une adresse sur 4 octets qui pointe sur une instruction. Si cette valeur n'est pas précisée, on devra utiliser la première entrée dans le code objet. Il n'y a aucune donnée.
longueur	2	Le nombre de paires de caractères de l'enregistrement, sans compter le *type* et la *longueur*.
adresse	4, 6 ou 8	L'adresse où doivent être rangées les données.
données	0 à 2n	de 0 à n octets. Une règle consiste à limiter la quantité de données par enregistrement à 28 octets (56 caractères).
somme de contrôle	2	Le *LSB* (pour *least significant byte*, en français octet le moins significatif) de la somme des données contenues dans l'enregistrement, en complément à 1. Pour vérifier, il suffit alors de faire la somme des données transmises + le somme de contrôle. Si le résultat est 0xFF, il n'y a pas eu d'erreur de transmission.

Le tableau 4.2 présente un exemple d'un fichier en format S-Record. Cet exemple contient :

- 1 enregistrement S0;
- 4 enregistrements S1;
- 1 enregistrement S9.

Tableau 4.2 Exemple d'un fichier en format S-Record

```
S00600004844521B
S1130000285F245F2212226A000424290008237C2A
S1130010000200080008262900185381234100181 3
S113002041E900084E42234300182342000824A952
S113003000144ED492
S9030000FC
```

Format Intel HEX

Le format Intel HEX est utilisé pour programmer les microcontrôleurs, les EEPROM et les autres composants programmables. Chaque ligne de format Intel HEX contient des valeurs hexadécimales avec une adresse absolue ou décalée. Le format peut être décrit comme un ensemble de lignes de texte. Chaque ligne, appelée aussi un enregistrement, respecte la syntaxe suivante: :BBAAAATTDDDD.....DDDDCC où:

1. ':' est le début de l'enregistrement;
2. BB est le nombre d'octets de l'enregistrement;
3. AAAA est l'adresse absolue (ou relative) du début de l'enregistrement;
4. TT est le champ spécifiant le type;
5. DD...DD est le champ des données;
6. CC est l'octet de somme de contrôle (*checksum*).

Il y a six valeurs possibles pour le champ de type:

1. 00, ce type indique que l'enregistrement contient des données et une adresse de 16 bit.
2. 01, ce type indique que l'enregistrement est la fin du fichier. Il n'y a pas de données après cet enregistrement. Il doit être la dernière ligne du fichier et donc cet enregistrement n'est permis qu'une seule fois par fichier, habituellement ':00000001FF'.
3. 02, ce type permet d'étendre l'adresse étendue. Il définit l'adresse de base du segment. Il est utilisé quand une adresse de 16 bits n'est pas suffisante. L'adresse spécifiée par le champ 02 est multipliée par 16 (décalée de 4 bits vers la gauche) et additionnée aux adresses contenues dans les champs de type 00. Ceci permet d'adresser 1 mégaoctet.
4. 03, *Start Segment Address Record*. Pour les processeurs 80x86, il spécifie le contenu initial des registres *CS: IP*. Le champ d'adresse est alors 0000. Le nombre d'octets est 04. Les deux premiers octets sont la valeur de *CS (Code Segment)*. Les deux suivants sont la valeur de *IP (Instruction Pointer)*.
5. 04, *Extended Linear Address Record*, ce type autorise un mode d'adressage sur 32 bits. Le champ d'adresse AAAA est 0000. Le champ nombre d'octets BB est 02. Les deux octets présents représentent les 16 bits de poids fort des adresses de 32 bits, quand ils sont combinés avec le type TT à 00.

6. 05, *Start Linear Address Record*. Le champ d'adresse AAAA est 0000, le champ nombre d'octets BB est 04. Les 4 octets de données représentent la valeur 32-bit chargée dans le registre du processeur.

Nous avons utilisé le compilateur WinARM (qui contient *GNU Compiler Collection*, abrégé en GCC) pour générer le fichier Intel HEX qui contient notre code (*firmware*) pour programmer le flash des microcontrôleurs. Nous avons développé un programmateur de flash (ROM) pour charger le microprogramme dans les microcontrôleurs.

Ce fichier Intel HEX contient le microprogramme de notre plateforme μC/MAS et particulièrement le code de l'agent mobile que nous avons précédemment présenté à la figure 4.5. Dans ce code en format Intel HEX, nous pouvons déterminer, par exemple, l'espace mémoire utilisé par le programme en octets, les adresses et les contenus des variables qui sont dans le segment de texte (code) comme *AddrNode1*, *AddrNode2* et *AddrNode3*, par exemple. Également, nous pouvons déterminer les adresses du début et de la fin du programme à l'aide de ces trois enregistrements extraits du code du fichier de la figure 4.16:

(1) :0400000580008000F7: Ceci permet de déterminer l'adresse du début du programme: 0x80008000;

(2) :02000004800278: Ceci permet de déterminer les 16 bits de poids fort de l'adresse de la fin du programme: 0x8002;

(3) :109E5800010000043000000000000000000000B6: Ceci permet de déterminer les 16 bits de poids faible de l'adresse de la fin du programme: 0x9E58 (l'adresse des 4 premiers octets)+0xC = 0x9E64.

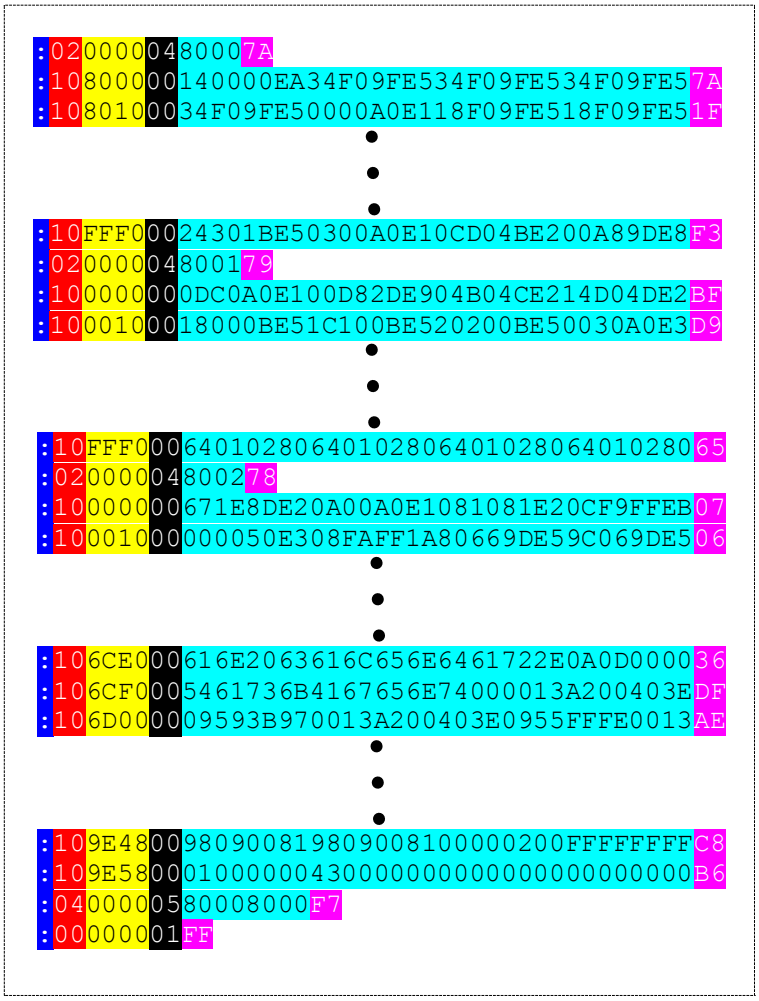


Figure 4.16 Fichier en format Intel HEX

L'adresse de la fin du programme peut être déterminée comme la somme de: 0x8002 0000 + 0x0000 9E64= 0x8002 9E64. Ainsi, nous pouvons calculer la taille du programme comme la différence des adresses de la fin et du début du programme

= 0x8002 9E64 - 0x8000 8000

= 0x21E64 octets;

= 138852 octets

$= \frac{138852}{1024} =$ 135 kilooctets.

Lors de la migration, le code exécutable de l'agent ne subit aucune transformation. Il est chargé dans la mémoire flash tel qu'il est produit par le compilateur. L'état d'exécution de l'agent est composé de la pile et du bloc de contrôle de la tâche *agent (Task Control Block,*

TCB). La pile contient les variables locales de la tâche *agent*. Les données courantes sont les variables globales exclusives à la tâche *agent*. Le TCB est une structure de données qui est utilisée par le noyau temps réel pour maintenir l'état de la tâche *agent* quand elle est préemptée. Le *TCB* contient le pointeur de pile, la priorité de la tâche *agent*, la taille de la pile, etc.

Contrairement au *TCB*, la pile et les données courantes sont d'abord encodées en format *Intel HEX* avant d'être incorporés dans le document XML. Le format XML présente la tâche *agent* (l'état d'exécution et les données courantes) comme un document et un analyseur syntaxique XML gère ce document. L'analyseur syntaxique structure le document, la façon de l'accéder et de le manipuler. Il offre les fonctionnalités suivantes: créer, naviguer, ajouter, modifier ou supprimer des éléments et leur contenu.

À l'aide de deux analyseurs/encodeurs (Intel HEX et XML) que nous avons conçus, il est possible de visualiser les représentations du contexte d'exécution d'une tâche a*gent* et des données globales à l'aide d'un éditeur XML. La figure 4.17 présente le contexte d'exécution visualisé à l'aide d'un éditeur XML.

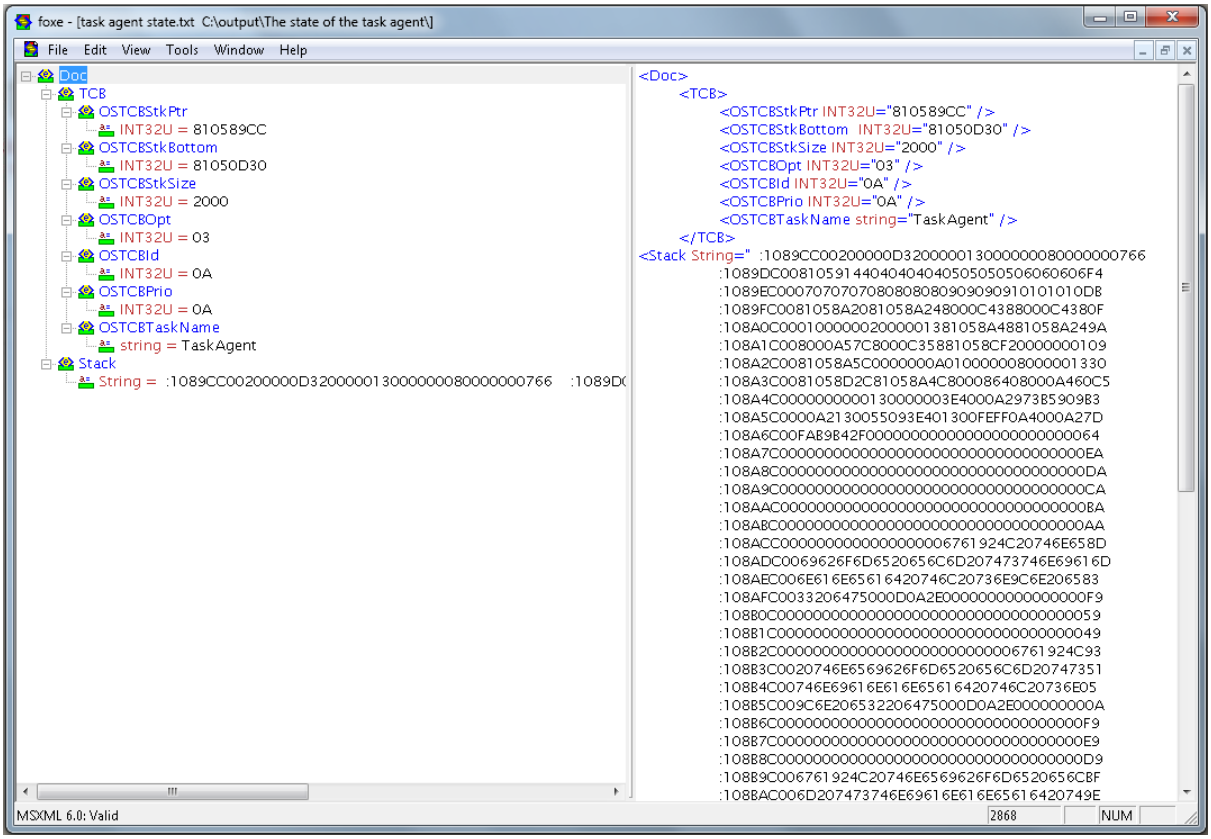


Figure 4.17 Contexte d'exécution visualisé par un éditeur XML

Le format *Intel HEX* permet la détection des erreurs lors de transfert d'enregistrements grâce à un système de somme de contrôle. Pour fin d'explication, prenons l'enregistrement suivant:
:1089CC00200000D32000001300000008000000766

La somme de contrôle de cet enregistrement peut être calculée comme la somme des octets en complément à 2. La somme des octets est:
10+89+CC+00+20+00+00+D3+20+00+00+13+00+00+00+08+00+00+00+07 = 29A
Le complément à 2 de 29A est: D66. On retient les 8 bits de poids faible, c'est-à-dire 66. Ceci est égal à la somme de contrôle de l'enregistrement.

La figure 4.18 présente l'algorithme de migration de tâche *agent* utilisé dans notre système. Notre algorithme de migration de tâche *agent* s'est inspiré de l'algorithme de copie totale. La migration de la tâche agent suit les étapes suivantes:

- La tâche *agent* envoie une demande de migration au destinataire;
- Le nœud destination peut répondre par un message d'acceptation ou de rejet;

- Si la migration est acceptée:
    - La tâche *agent* provoque un changement de contexte d'exécution;
    - Le nœud destinataire envoie une demande de transfert de code au serveur;
    - Le serveur transfère le code au nœud destinataire;
    - Le nœud destinataire informe au nœud source qu'il a reçu le code;
    - Le nœud source transfère les données migrantes au destinataire;
    - Le nœud source transfère le contexte d'exécution;
    - Le nœud source informe le destinataire qu'il a fini le transfert;
    - La tâche agent reprend son exécution sur le nœud destinataire;
    - Le nœud source supprime la tâche agent.

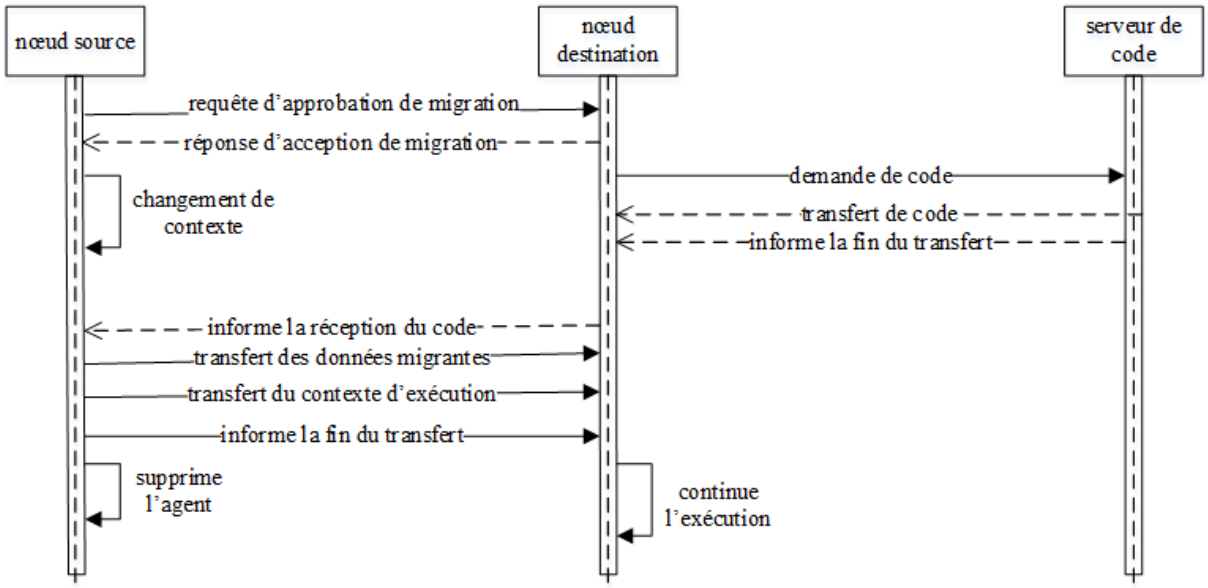


Figure 4.18 Algorithme de migration d'une tâche agent

Comme nous le montre la figure 4.19, chaque nœud de la plateforme µC/MAS est composé d'un module:

- de chargeur de code;
- d'environnement d'exécution d'agents mobiles:
    - Le service de capture et de transfert;
    - Le service réception et de restauration.
- de noyau temps réel;
- d'analyseur/encodeur Intel HEX;

- d'analyseur/encodeur XML;
- de réseau de communication.

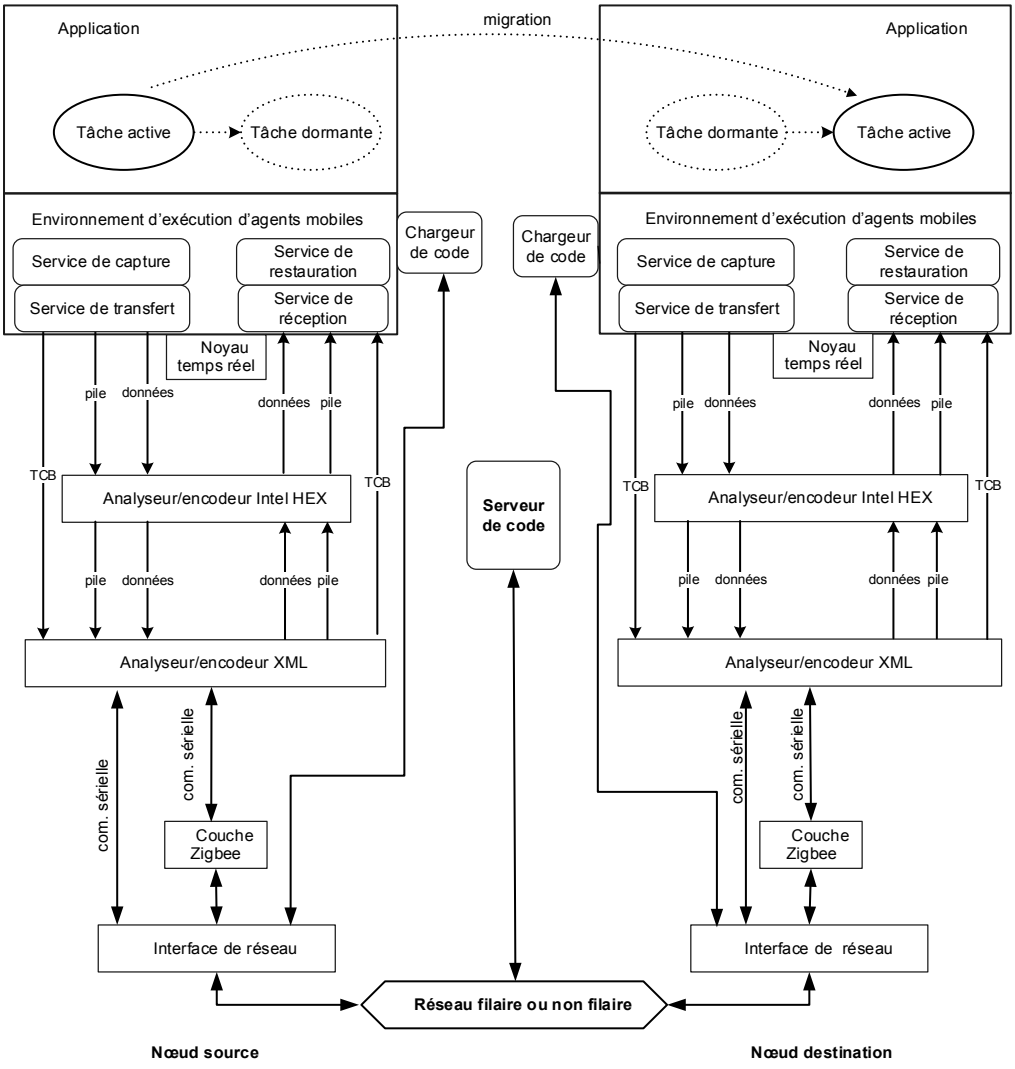


Figure 4.19 Interconnexion d'un nœud source et d'un nœud destination

Chargeur de code de la tâche *agent*

Comme nous travaillons dans un environnement de ressources limitées, notre système n'a pas de chargeur de code comme il y en a dans la machine virtuelle Java ou les autres systèmes. Alors, nous avons développé un chargeur de code. En effet, c'est un programmateur de mémoire flash (ROM) des microcontrôleurs. Il réside dans la mémoire interne du

microcontrôleur du nœud et veille l'arrivée des agents mobiles. Lorsqu'un agent demande la migration, il charge le code depuis le serveur et programme la mémoire flash externe du microcontrôleur.

Service de capture et de transfert de la tâche *agent*

Il existe deux types de services service de capture et de transfert de la tâche *agent*. Le premier est la capture et le transfert des blocs des données migrantes de la tâche agent. Ce service consiste à extraire chaque bloc les valeurs des données, puis il les met dans une représentation en format Intel HEX en combinaison avec XML à l'aide de deux encodeurs imbriqués conçus pour cette fin. Ensuite, la représentation est transférée vers le nœud de destination.

Le second est la capture et le transfert du contexte d'exécution, c'est-à-dire la pile et le bloc de contrôle de la tâche. Ceci consiste à extraire le contexte d'exécution des données et les adresses. Puis, les données et les adresses du contexte d'exécution sont encodées dans une représentation en format Intel HEX en combinaison avec XML. Ensuite, la représentation est envoyée vers le nœud de destination via le réseau. Enfin, sur le nœud source, la tâche agent est supprimée.

Service de réception et de restauration de la tâche *agent*

Il existe deux types de services de réception et de restauration de la tâche agent. Le premier est la réception et la restauration des blocs des données migrantes de la tâche agent. Une fois que la représentation des blocs des données migrantes est reçue, les analyseurs XML et Intel HEX extraient les valeurs. Puis, les blocs des données migrantes sont restaurés dans le nœud de destination.

Le second est la réception et la restauration du contexte d'exécution. Ceci consiste à extraire le contexte d'exécution les données et les adresses. Une fois que la représentation du contexte d'exécution est reçue, les analyseurs XML et Intel HEX extraient les valeurs et les adresses. Puis, une nouvelle tâche agent est créée avec le contexte d'exécution reçu dans le nœud de destination. La nouvelle tâche continue son exécution là où elle a été arrêtée.

4.3 Conclusion

Dans ce chapitre, nous avons présenté les différents éléments de notre solution de la mobilité d'agents en exploitant l'analogie qui existe entre le changement du contexte d'exécution de tâches et la mobilité d'agents. Notre solution de la mobilité d'agents se traduit concrètement par le développement d'une plateforme pour systèmes embarqués temps réel qui repose sur l'extension d'un noyau temps réel. La solution de la mobilité d'agents est subdivisée en trois éléments: une méthode de migration, une directive et un format de transfert d'agents.

Notre méthode de migration d'agents est basée sur l'extension des services de bas niveau d'un noyau temps réel. En effet, nous avons étendu le noyau temps réel en ajoutant des mécanismes permettant de: capturer, transférer et restaurer le contexte d'exécution de tâche *agent*. Ces mécanismes sont intégrés aux fonctionnalités du noyau temps réel permettant ainsi la mobilité d'agents logiciels au sein d'une grappe de systèmes embarqués.

La directive de migration d'agents permet d'optimiser les ressources (espace mémoire, temps de transfert, etc.) en spécifiant les données essentielles qui migrent avec l'agent. Ainsi, les développeurs d'application peuvent choisir les données essentielles qui migrent avec l'agent en les plaçant dans les segments de mémoire appropriés. En effet, ils peuvent définir les données essentielles dans des variables locales et/ou dans des blocs des données migrantes.

Nous avons conçu un format de transfert permettant d'intégrer de façon élégante les différents composants de la tâche *agent* afin de construire une représentation de l'état d'exécution. Cette représentation est transmise au nœud de destination via le réseau pour que la tâche *agent* reprenne son exécution là elle a été interrompue avant sa migration.

La spécificité de notre solution de la mobilité d'agents est qu'elle peut être mise en œuvre dans n'importe quel système multitâche puisque nous exploitons des mécanismes de bas niveau qui existent dans ces systèmes. Le principal avantage de notre solution de la mobilité d'agents, qui se matérialise par une plateforme d'agents mobiles, est sa petite taille et son intégration dans l'environnement pour systèmes temps réel. En outre, notre plateforme d'agents mobiles

supporte la mobilité forte. En effet, à notre connaissance, il n'existe pas de plateformes d'agents mobiles permettant la mobilité forte dans le contexte des systèmes temps réel utilisant une mémoire aussi petite qu'un mégaoctet (RAM et ROM).

CHAPITRE 5
MISE EN ŒUVRE DE LA SOLUTION DE MOBILITÉ D'AGENTS PROPOSÉE

Dans le chapitre précédent, nous avons présenté l'architecture des différents éléments constituant la solution de mobilité d'agents proposée. Dans ce chapitre, nous exposons la mise en œuvre de la solution de mobilité d'agents. Premièrement, nous décrivons l'architecture de la bibliothèque de la plateforme µC/MAS. Deuxièmement, nous présentons les primitives permettant la capture et la restauration du contexte d'exécution. Puis, nous décrivons les fonctions permettant la capture et la restauration des données courantes. Ensuite, nous exposons les services de transport utilisés par l'agent mobile. Enfin, nous exposons l'évaluation de la plateforme d'agents mobiles en effectuant un certain nombre des tests sur différents réseaux.

5.1 Introduction

Pour passer du changement de contexte d'exécution à la migration de l'agent, nous étendons les mécanismes de bas niveau de suspension, de sauvegarde et de reprise de tâche qui existent déjà dans les noyaux temps réel. La mise en œuvre de notre solution de mobilité proposée a abouti la plateforme µC/MAS. Celle-ci est conçue sur la base d'un noyau temps réel appelé µC/OS-II. Ce dernier est dédié pour opérer sur des machines avec des ressources très limitées (taille de mémoire, sans unité de gestion de mémoire, etc.). Les caractéristiques du noyau µC/OS-II sont [Labrosse, 2002]:

- Multitâches et préemptif: exécute la tâche la plus prioritaire qui est prête;
- Déterministe;
- Fiable et robuste;
- Portable;
- ROMable: peut être mis en mémoire ROM;

- Faible empreinte mémoire (24 Ko maximum);
- Nombre de tâches maximum: 250;
- Configurable pour n'offrir que les fonctionnalités requises;
- Indépendance de la plateforme utilisée.

5.2 Architecture de la bibliothèque de la plateforme µC/MAS

La figure 5.1 présente une vue globale de la plateforme d'agents mobiles µC/MAS. Cette plateforme est mise en œuvre sous forme d'une bibliothèque de programmes. Cette bibliothèque est conçue pour s'intégrer aux fonctionnalités d'un noyau temps réel existant, en occurrence µC/OS-II, mais aussi potentiellement d'autres de mêmes types telles que µC/OS-III ou µClinux. Elle est essentiellement destinée aux environnements de très petite taille construits autour de microcontrôleurs.

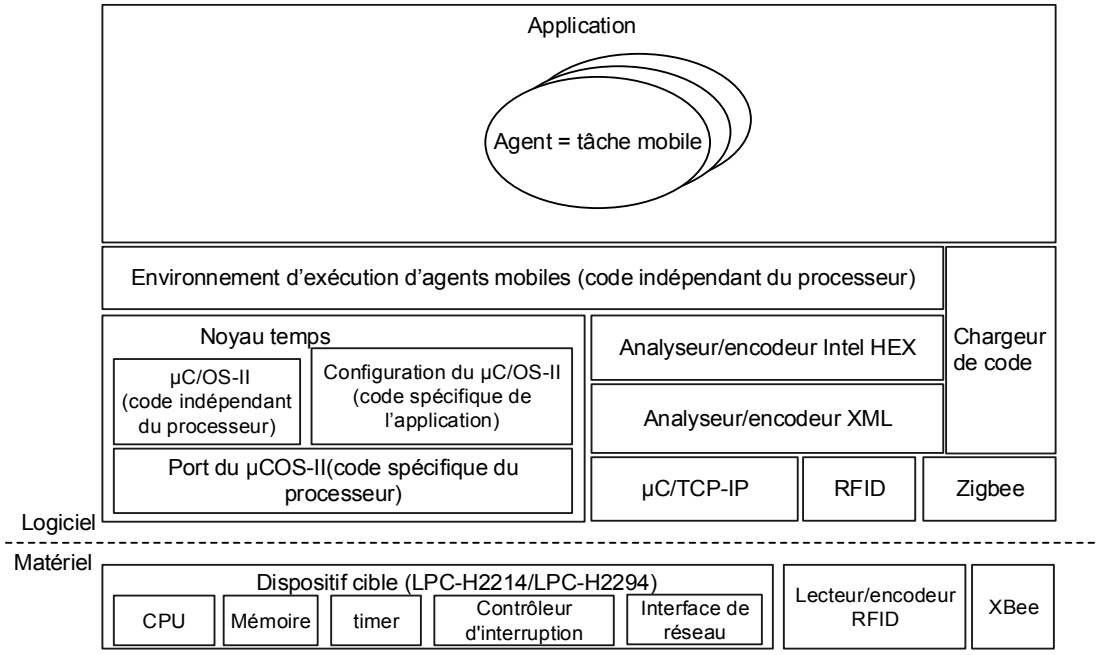


Figure 5.1 Vue générale de la plateforme µC/MAS

Le noyau choisi, en l'occurrence µC/OS-II, doit être écrit en C ANSI et son code source doit être disponible afin de permettre l'accès aux mécanismes de capture et de restauration du contexte d'exécution qui vont être exploités par la plateforme d'agents mobiles.

La figure 5.2 illustre l'architecture de la bibliothèque de la plateforme µC/MAS. La bibliothèque de la plateforme d'agents mobiles est construite selon une architecture modulaire qui permet son intégration avec d'autres services en utilisant un noyau temps réel et différents moyens de transport pour les agents. Elle comporte les modules suivants à la base de son intégrabilité avec ces services:

- Un module de capture/restauration qui assure l'interface avec le noyau temps réel;
- Un module d'encodage/décodage qui permet une mise en forme de données courantes selon le mode de transport utilisé;
- Un module d'encodage/décodage qui permet une mise en forme du contexte de l'agent selon le mode de transport utilisé;
- Un module offrant différents services de transport qui assurent l'interface avec les moyens de transport utilisés. Ces derniers peuvent être synchrones (comme dans le cas d'une liaison série, d'un réseau ZigBee) ou asynchrones (comme dans le cas de l'utilisation d'une carte intelligente de stockage RFID).

Le mécanisme de migration des agents fonctionne sommairement comme suit. Lorsque l'agent est présent sur un nœud, les données courantes sont mises dans des blocs de partitions et le fil d'exécution est hébergé dans une tâche *agent* (au sens du noyau utilisé). Lorsque l'agent décide de migrer vers un autre nœud, dans un premier temps, son code est chargé depuis un serveur.

Dans un deuxième temps, les données courantes sont envoyées vers le nœud de destination en utilisant le moyen de transport choisi (TCP/IP, ZigBee, RFID, etc.). Au nœud de destination, un mécanisme de veille est à l'écoute et identifie le transport utilisé pour recevoir les données qui arrivent. Sur le nœud de destination, les données sont reconstruites dans le même type de blocs de partition que sur le nœud source par une tâche *serveur* qui fait partie du système d'agents.

Dans un troisième temps, la tâche *agent* sur le nœud source est suspendue mais son contexte est sauvegardé. Le contexte de la tâche *agent* est transporté vers le nœud de destination. Au nœud de destination, le contexte est reçu par la tâche *serveur*. Ce contexte arrivant est utilisé

par la tâche *serveur* (par restauration) pour créer une nouvelle tâche qui va héberger la suite du fil d'exécution de cet agent.

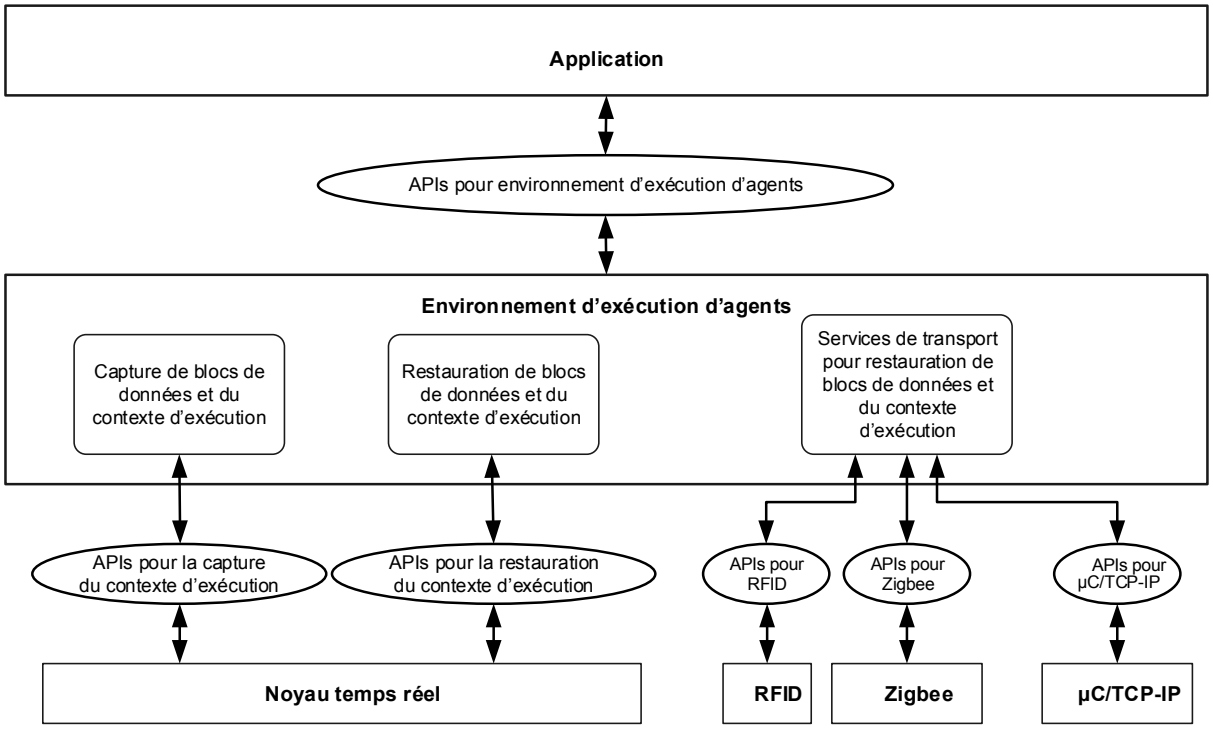


Figure 5.2 Architecture de la bibliothèque de la plateforme d'agents mobiles

5.2.1 Capture et restauration du contexte d'exécution

Pour capturer le contexte d'exécution d'une tâche *agent*, il faut d'abord provoquer un changement du contexte d'exécution de l'agent. Pour ce faire, nous avons mis en œuvre la primitive *OSTaskMoveTo()*. Cette dernière est appelée par la tâche *agent* lorsqu'elle décide de migrer. L'appel de cette primitive *OSTaskMoveTo()* provoque un changement de contexte de la tâche *agent*. Cette primitive a comme paramètre les variables *AddrNode, MediumType* et *err*. La variable *AddrNode* contient l'adresse du mécanisme de transport utilisé comme par exemple une adresse IP, une adresse MAC (*Media Access Control address*) pour un réseau Zigbee ou un identificateur d'une carte à puce RFID. La variable *MediumType* définit le type de protocole de transport utilisé: TCP/IP, Zigbee, carte à puce RFID, etc. Pour vérifier l'état du transfert, la variable *err* est utilisée et elle peut prendre une des valeurs suivantes:

- OS_NO_ERR: le transfert de la tâche agent est réussi;

- OS_PRIO_INVALID: la priorité associée à l'agent n'est pas disponible sur le nœud destination;
- OS_ADDR_INVALID: l'adresse de destination n'est pas valide;
- OS_TASK_TIME_OUT: le nœud de destination ne répond pas;
- OS_MIGR_ERR: une erreur s'est produite pendant l'opération de transfert.

Dans les cas où la migration échoue, l'agent poursuit son exécution dans le nœud où il se trouve.

Une fois le changement du contexte complété, la *tâche serveur* (comme décrit précédemment) prend le contrôle du processeur. Elle appelle la fonction *OSTskCtextCapture()* qui capture et encode la pile, et le bloc de contrôle de tâche *agent* dans un format (la représentation) approprié pour le transport utilisé. Pour capturer la pile, la plateforme utilise les valeurs sauvegardées dans le bloc de contrôle de tâche (voir la figure 5.3 qui illustre la mise en œuvre de la structure interne de l'agent):

- *OSTCBStkPtr*: le pointeur du sommet de la pile (dernière position utilisée dans la pile);
- *OSTCBStkBottom:* le pointeur vers le bas de la pile;
- OSTCBStkSize: la taille de la pile.

À l'aide de ces trois valeurs, l'espace mémoire utilisé par la pile, dans le cas d'une pile croissante vers les adresses (de 4 octets) les plus basses, peut être calculé comme suit: *OSTCBStkBottom* + 4 * *OSTCBStkSize* – *OSTCBStkPtr*. L'espace mémoire utilisé et le pointeur de la pile étant connus, on peut extraire les adresses et les données. Ensuite, les adresses et les données extraites de la pile et le bloc de contrôle de tâche sont encodés dans un format approprié pour le transfert, c'est-à-dire la combinaison des formats *Intel HEX* et *XML*.

Une fois que le contexte d'exécution de l'agent est encodé (sous forme d'une représentation), la fonction *OSTkCtextTransfert()* est appelée pour le transférer vers le nœud de destination. Cette opération de transfert est transparente pour l'utilisateur.

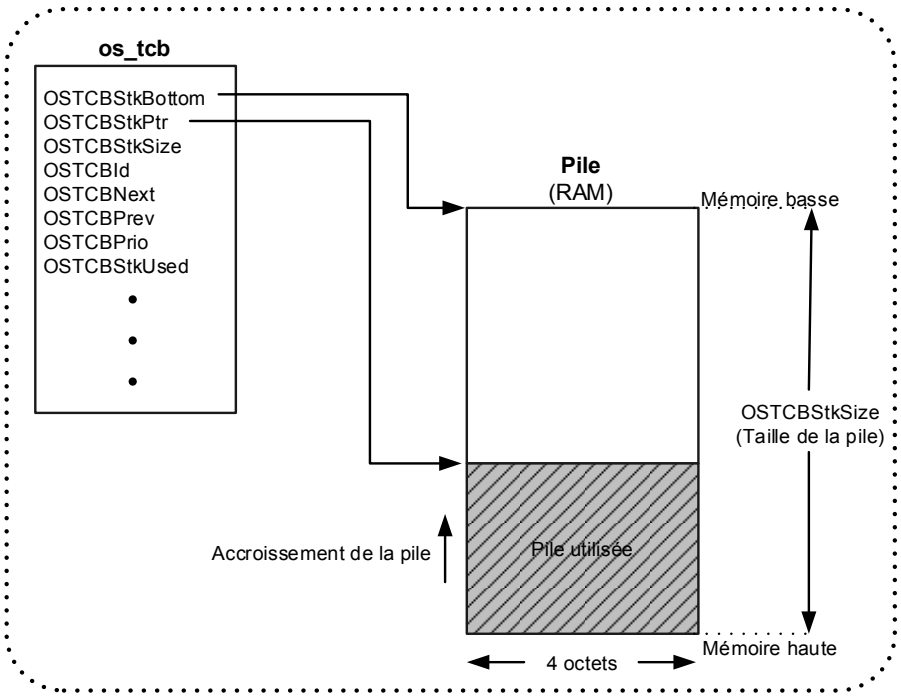


Figure 5.3 Structure interne du contexte d'exécution de l'agent

Dans le nœud de destination, la représentation du contexte d'exécution de l'agent est reçue par la tâche *serveur*. Cette dernière décode la représentation pour en extraire la pile et le bloc de contrôle de la tâche, puis elle la restaure dans une nouvelle tâche. Pour ce faire, nous nous sommes inspiré de la fonction *OSTaskCreateExt()* du noyau temps réel *µC/OS-II* pour mettre œuvre la primitive *OSTskCrteWihState()*. Ainsi, la description des arguments de la fonction *OSTaskCreateExt()* s'impose avant de présenter ceux de la primitive *OSTskCrteWihState()*.

La fonction *OSTaskCreateExt()* a les neuf arguments suivants:

(1) *task* est un pointeur vers le code de la tâche*;*

(2) *p_arg* est un pointeur vers une zone de données optionnel qui peut être utilisé pour transmettre des paramètres de la tâche lorsque la tâche est exécutée en premier.

(3) *ptos:* Une constante appelée *OS_STK_GROWTH* est utilisée pour configurer le sens de l'accroissement de pile de la tâche. Lorsque la constante *OS_STK_GROWTH* est mise à 1, la pile croît vers la mémoire basse.

(4) *prio* est un nombre entier qui indique la priorité de la tâche. Une priorité unique doit être assignée à chaque tâche. Plus le nombre est bas, plus la priorité est élevée.

(5) *id* indique l'identité de la tâche. *id* prend un nombre entre 0 et 65535.

(6) *pbos* est un pointeur sur le bas de pile de la tâche.

(7) *stk_size* indique le nombre d'éléments (en octets) dans la pile.

(8) *pext* est un pointeur vers un emplacement de mémoire qui est utilisé comme une extension du bloc de contrôle de tâche. Par exemple, cette mémoire peut contenir le contenu des registres à virgule flottante pendant un changement de contexte, le temps de chaque tâche nécessaire à l'exécution, le nombre de fois que la tâche a été commutée, etc.

(9) *opt* contient des informations supplémentaires (ou optionnelles) sur le comportement de la tâche. Les huit 8 bits inférieurs sont réservés par µC/OS-II tandis que les huit 8 bits supérieurs peuvent être spécifique à l'application. Les choix actuels sont:

- OS_TASK_OPT_STK_CHK: Vérifier la pile allouée à la tâche;
- OS_TASK_OPT_STK_CLR: Initialiser à zéro lors de la création de la tâche;
- OS_TASK_OPT_SAVE_FP: Si le processeur dispose de registres à virgule flottante, il faut les sauvegarder lors du changement de contexte.

La fonction *OSTaskCreateExt()* retourne une des valeurs suivantes:

- OS_NO_ERR si la création de la fonction est réussie;
- OS_PRIO_EXIT si la priorité de la tâche existe déjà;
- OS_PRIO_INVALID si la priorité n'est pas valide;
- OS_ERR_TASK_CREATE_ISR si on a essayé de créer une tâche à partir d'un ISR.

La figure 5.4 présente la fonction de création de tâches *OSTaskCreateExt()* du noyau temps réel µC/OS-II.

```
INT8U  OSTaskCreateExt (void   (*task)(void *pd),
                        void    *p_arg,
                        OS_STK  *ptos,
                        INT8U    prio,
                        INT16U   id,
                        OS_STK  *pbos,
                        INT32U   stk_size,
                        void    *pext,
                        INT16U   opt)
```

Figure 5.4 Fonction de création de tâches *OSTaskCreateExt()* du noyau temps réel *µC/OS-II*.

Comme illustrée à la figure 5.5, la primitive *OSTskCrteWihState()* a dix arguments, donc un paramètre de plus que la fonction *OSTaskCreateExt()* offerte par le noyau µC/OS-II. Cet argument additionnel représente l'état initial de la tâche. À l'aide de cet argument, la tâche *agent* est restaurée à la destination. Pour ce faire, la primitive *OSTskCrteWihState()* fait appel à la fonction *OSTskCntextRstore()*. Sur le nœud de destination, les données présentes dans la pile de la tâche *agent* sont identiques à celles qui étaient sur le nœud source. Ainsi, la tâche *agent* reprend donc son exécution là où elle avait été interrompue sur le nœud source.

Dans la plateforme µC/MAS, les utilisateurs ont deux fonctions pour la création d'une tâche *agent*:

1. La primitive *OSTaskCreateExt()* du noyau temps réel µC/OS-II qui permet de créer de la tâche *agent* sans contexte initial;
2. La primitive *OSTskCrteWihState()* qui permet de créer une *tâche agent* avec un état initial pour que l'exécution reprenne le point où elle a été arrêtée.

```
INT8U  OSTskCrteWihState (void   (*task)(void *pd),
                          void    *p_arg,
                          OS_STK  *ptos,
                          INT8U    prio,
                          INT16U   id,
                          OS_STK  *pbos,
                          INT32U   stk_size,
                          void    *pext,
                          INT16U   opt,
                          mxml_node_t  *tree)
```

Figure 5.5 Fonction de création de tâches *OSTskCrteWihState()* avec un état initial

La figure 5.6 illustre la représentation d'un contexte d'exécution en (a) versus sa restauration de la pile en (b). Notons que la norme Intel HEX peut encoder des adresses de 8, 16 ou 32 bits. Bien que les modules LPC-H2214/LPC-H2294 utilisent des adresses de 32 bits, nous encodons la pile en format de 16 bits pour minimiser l'espace de stockage ainsi que le temps de transfert.

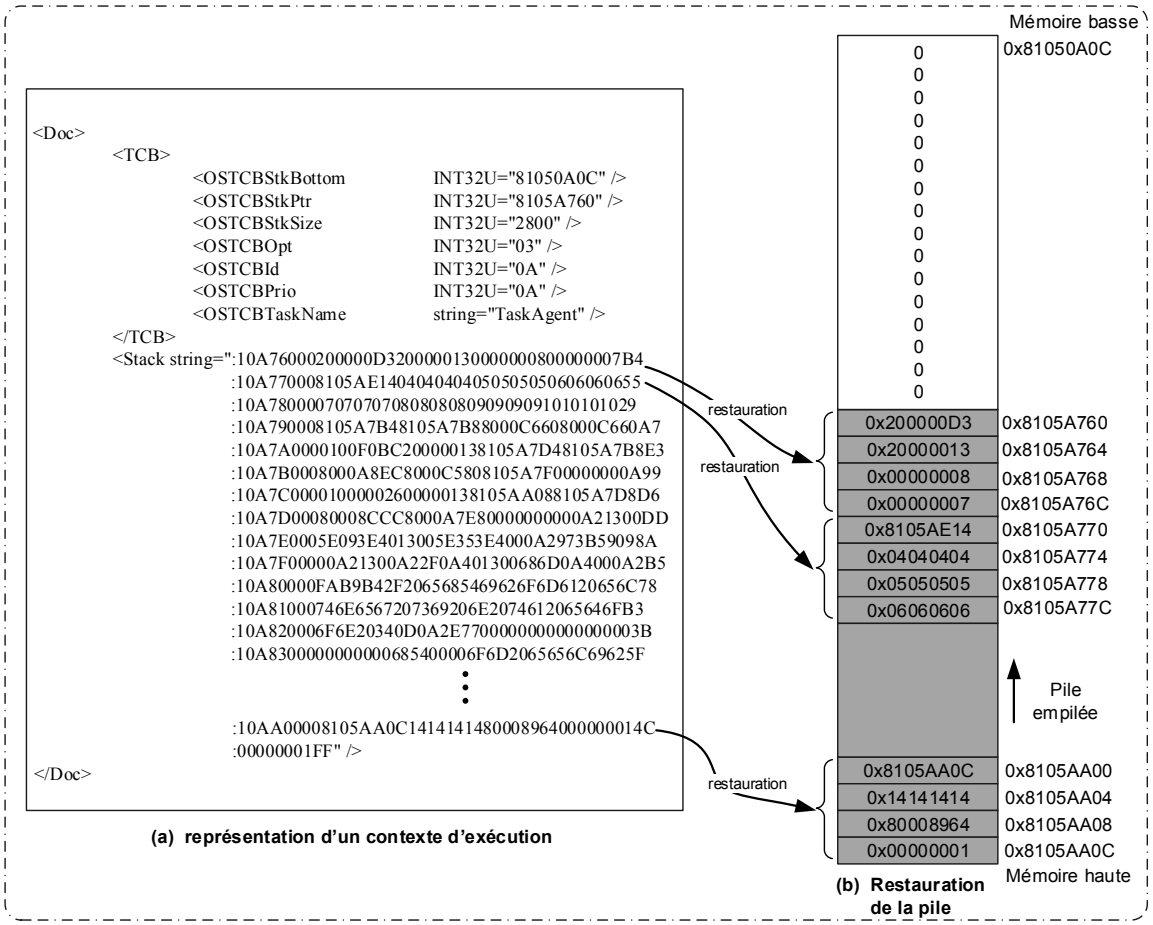


Figure 5.6 Représentation d'un contexte d'exécution en (a) versus la restauration de la pile en (b)

À titre d'exemple, prenons le cas de la deuxième ligne du contenu de la pile (voir la figure 5.6 (a)). Les caractères A770 représentent les 16 bits les moins significatifs. L'adresse absolue peut être déterminée comme suit: 0x8105 (16 bits les plus significatifs) + 0xA770 (le décalage) = 0x8105 A770.

5.2.2 Capture et restauration des données courantes

Rappelons qu'une tâche *agent* est constituée d'un code, un contexte d'exécution et des données courantes. Pour la migration des données globales, des partitions de blocs de mémoire offertes par le noyau µC/OS-II peuvent être étendues afin de concevoir des mécanismes de transfert. Lors de la migration d'un bloc de données, la tâche *agent* fait appel à la primitive *OSMemBlckMove()*. Celle-ci permet d'extraire les données du bloc de mémoire pour les sauvegarder, puis les encoder dans un format approprié selon le transport utilisé.

La figure 5.7 présente la séquence d'utilisation de deux primitives: *OSMemBlckMove()* et *OSTaskMoveTo()*. En effet, comme nous présente cette figure, le bloc des données est transféré en premier à l'aide de la primitive *OSMemBlckMove()*. Les paramètres de la primitive *OSMemBlckMove()* sont:

- *DataBlocPtr* qui pointe le bloc de données à transférer vers le nœud de destination;
- *DestNodeAddr* qui contient l'adresse réseau de nœud de destination;
- *DestDataBlocAddr* qui retourne l'adresse du pointeur de bloc de données de nœud de destination;
- *err* qui retourne une des valeurs suivantes:
    - OS_NO_ERR: le transfert de bloc des données migrantes est réussi;
    - OS_ADDR_INVALID: l'adresse de destination n'est pas valide;
    - OS_TASK_TIME_OUT: le nœud de destination ne répond pas;
    - OS_MIGR_ERR: une erreur s'est produite pendant l'opération de transfert.

Pour effectuer la migration de blocs des données, la primitive *OSMemBlckMove()* fait appel à la fonction *OSMemBlkCapture()*. Cette fonction capture les blocs de données. Puis, la fonction *OSMemBlkCapture()* met les blocs de données en format de transfert à l'aide deux analyseurs/encodeurs imbriqués que nous avons conçus pour cette fin. Ensuite, la primitive *OSMemBlkTransfert()* transfère les blocs de données vers le nœud de destination. Enfin, la primitive *OSMemBlckMove()* libère le bloc de données du nœud source si le transfert est réussi. Les fonctions *OSMemBlkCapture()* et *OSMemBlkTransfert()* sont transparentes pour l'utilisateur.

```
static  void  TaskAgent (void *p_arg) /*L'agent commence son exécution sur le nœud AddrNode1*/
{
        INT8U err, MediumType = 1;
        INT8S string[110], message[100];
        INT32S i, a = 10, b = 30, c = 55, d = 2013, somme = 0;
        INT8U AddrNode1[10] = {0x00, 0x13, 0xA2, 0x00, 0x40, 0x0A, 0x2F, 0xB4, 0xB9, 0xFA};
        INT8U AddrNode2[10] = {0x00, 0x13, 0xA2, 0x00, 0x40, 0x3E, 0x09, 0x55, 0xFF, 0xFE};
        INT8U AddrNode3[10] = {0x00, 0x13, 0xA2, 0x00, 0x40, 0x3E, 0x09, 0x59, 0x3B, 0x97};
        INT32U *pBlockDest;

        somme += a + b + c;
        for(i = 0; i < 32; i++) pBlockSource [i] =0;/*Initialer le bloc pBlockSource à 0*/
        for(i = 0; i < 32; i++) pBlockSource [i] += i;/*Mettre des données dans le bloc pBlockSource*/


        for(i = 0; i < 32; i++) {
          sprintf(string, "%04X\n\r",pBlockSource[i]);/*Copier le contenu de chaque élément du pBlockSource dans  string*/
          UART0WriteTxt(string);//Afficher le contenu de la variable string dans l'UART 0*/
        }

        OSMemBlckMove(pBlockSource, &pBlockDest, AddrNode2, &err);/*Le transfert du bloc pBlockSource*/
        if(err != OS_NO_ERR){
                sprintf(string, "La migration du bloc des données est échouée, le type d'erreur est: %d\n\r",err);
                UART0WriteTxt(string);//Afficher le message d'erreur*/
          }

        OSTaskMoveTo(AddrNode2, MediumType, &err); /*La migration du contexte d'exécution*/
        if(err != OS_NO_ERR){
                sprintf(string, "La migration du contexte d'exécution est échouée, le type d'erreur est: %d\n\r",err);
                UART0WriteTxt(string);//Afficher le message d'erreur*/
        }

        somme += a + b + c; /*Ici, l'agent arrive dans le nœud AddrNode2 */
        sprintf(string, "somme = %08X \n\r", somme); /*Copier le contenu de la variable somme dans  string*/
        UART0WriteTxt(string);/*Afficher le contenu de la variable string sur le port UART de l'ordinateur*/
        for(i = 0; i < 32; i++) {
          sprintf(string, "%04X\n\r", pBlockSource[i]);/*Copier le contenu de chaque élément du pBlockSource dans  string*/
          UART0WriteTxt(string);/*Afficher le contenu de la variable string sur le port UART de l'ordinateur*/
        }
        •
        •
        •
       while(1){}
}
```

Figure 5.7 Séquence d'utilisation de primitives *OSMemBlckMove()* et *OSTaskMoveTo()*

La figure 5.8 montre le modèle de format de transfert de blocs des données migrantes. Sur le nœud destinataire, le bloc des données migrantes est reçu par la tâche de veille qui joue le rôle d'un serveur d'agents. Cette tâche décode et reconstruit le bloc de données dans le même type de partition de mémoire que celui du nœud source. Pour ce faire, nous avons mis en œuvre la fonction *OSMemBlkRestore()*. La restauration d'un bloc de données se déroule de la façon suivante: un bloc de donnée de même type et de même taille que celui du nœud source est créé dans le nœud de destination. Ensuite, les données reçues du nœud source sont copiées dans le

bloc nouvellement créé à la destination. Puis, l'adresse du bloc et un code indiquant que la restauration a réussi sont envoyés au nœud source. Enfin, sur le nœud source, l'adresse du bloc de données reçue est copiée dans une variable locale, en l'occurrence dans *DestDataBlocAddr*, pour utiliser cette variable lorsque la tâche *agent* arrive à la destination. La fonction *OSMemBlkRestore()* est transparente pour l'utilisateur.

```
<Bloc>
        <BlocSize>value</BlocSize>
        <BlocIntelHex =":XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                       :XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                              •
                              •
                              •
                       :00000001FF" />

</Bloc>
```

Figure 5.8 Modèle de format de transfert de blocs des données migrantes

Afin de mettre les blocs de données migrants et le contexte d'exécution de la tâche *agent* dans un format de transfert, nous avons mis en œuvre deux encodeurs/analyseurs imbriqués. Essentiellement, l'analyseur/encodeur Intel HEX est composé les trois fonctions suivantes:

- *IntelHexFormat()*: Cette fonction est utilisée par les primitives *OSTskCtextCapture()* et *OSMemBlkCapture()* afin d'encoder les blocs de données migrants et le contexte d'exécution en format Intel HEX.
- *ParseHexLine()*: Cette fonction analyse et valide chaque enregistrement du format Intel HEX. Celle-ci est utilisée par la fonction *IntelHexFormat()*.
- *CheckSum()*: Cette fonction permet de calculer la somme de contrôle de chaque enregistrement.

Afin de mettre en œuvre l'analyseur/encodeur XML, nous avons modifié Mini-XML [Sweet, 2011] qui était initialement conçu pour les systèmes à l'usage général afin de l'adapter à notre plateforme d'agents mobiles. En effet, dans le contexte de systèmes embarqués, l'espace mémoire est limité. Par conséquent, nous avons modifié le Mini-XML afin d'extraire les fonctionnalités qui ne sont pas nécessaires dans notre plateforme pour économiser de l'espace mémoire. Ainsi, notre analyseur/encodeur XML fournit les fonctionnalités suivantes:

- Le stockage des données dans une structure arborescente de liste chaînée en préservant la hiérarchie des données XML;
- Le support des noms des éléments, leurs attributs et les valeurs des attributs sans limites prédéfinies autre que la mémoire disponible;
- Le support de type d'entier, de réel, de texte, etc.;
- Les fonctions pour créer et gérer les arbres de données.
- Les fonctions pour trouver et se déplacer pour repérer et naviguer facilement dans les arborescences de données.

5.3 Services de transport

Lors de migration, l'agent peut utiliser un de trois types de services de transport: filaire (liaison série, µC/TCP-IP) ou sans fil (Zigbee, des cartes à puce sans contact). Pour réaliser ces services, une couche adaptable est mise en œuvre. Cette couche assure l'interface entre les protocoles et les dispositifs des périphériques.

5.3.1 Réseau Zigbee

ZigBee est un protocole de haut niveau permettant la communication de petites radios, à consommation réduite, basée sur la norme IEEE 802.15.4 pour les réseaux à dimension personnelle (Wireless Personal Area Networks : WPAN). Le 802.15.4 est un protocole de communication défini par IEEE *(Institute for Electrical and Electronics Engineers)*. Il est destiné aux réseaux sans fil de la famille des *LR WPAN (Low Rate Wireless Personal Area Network)* du fait de leur faible consommation, de leur faible portée et du faible débit des dispositifs utilisant ce protocole.

ZigBee est caractérisé par une portée comprise entre 10 et 100 mètres et un débit maximum de 250 kbits/s. ZigBee utilise la couche physique et la couche liaison définie par le standard IEEE 802.15.4, qui sont les couches 1 et 2 du modèle OSI ainsi que le couche réseau et applicatif développés par ZigBee Alliance. Dans ce contexte, chaque nœud (module) doit s'identifier avant de rejoindre le réseau. Un réseau est constitué des modules suivants:

- Un *nœud coordonnateur* gère les fonctions de haut niveau du réseau comme l'authentification, la sécurité, etc. Un seul coordonnateur doit être présent dans chaque réseau (ensemble de nœuds partageant un même identifiant de réseau *PAN*, pour *Personal Area Network*).
- Des *nœuds routeurs* disposent de toutes les fonctions d'un *nœud final* avec en plus des fonctions de haut niveau utiles pour étendre le réseau. Ils permettent:
    - d'étendre la taille du réseau en permettant aux autres modules de s'enregistrer auprès d'eux et non exclusivement auprès du coordonnateur. Cela évite de saturer le coordonnateur.
    - d'étendre la portée du réseau. Chaque module routeur répète les signaux reçu aux autres modules qui lui sont enregistrés. Le signal se répercute ainsi de module en module pour atteindre le *nœud final* concerné.
- Des *nœuds finals* ne sont actifs que sur changement de leurs entrées/sorties ou sur réponse à une trame, leur consommation est donc très faible.

La figure 5.9 présente la pile de Zigbee. Pour faire communiquer deux modules, il existe deux modes d'utilisation:

- Un mode "transparent" où les données sont directement envoyées/reçues par le module. La configuration du module s'effectue par le biais de commande "AT". Lorsque le réseau Zigbee est configuré en mode "transparent", le message envoyé par un nœud est reçu non seulement par le destinataire mais également par tous les autres nœuds.
- Un mode "API" dans lequel il faut fabriquer ses propres trames et les envoyer au module. Ce mode est plus puissant et rapide mais nécessite d'écrire un code pour envoyer et recevoir les trames. Dans le cadre de ce projet, nous avons utilisé le mode API. Lorsque le réseau Zigbee est configuré en mode "API", le message envoyé par un nœud source est reçu seulement par le destinataire grâce à son adresse MAC incorporée dans le message.

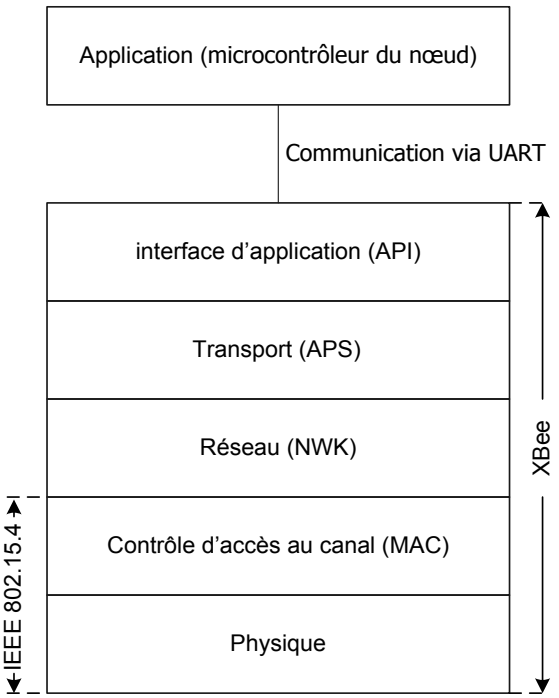


Figure 5.9 Pile de Zigbee

Afin de déployer un réseau ZigBee [ZigBee Standards Organization, 2008], nous avons utilisé des modules XBee. Ces modules, interfacés au microcontrôleur du nœud via un port série, mettent en œuvre la pile ZigBee sur leur propre microcontrôleur dédié. La figure 5.10 montre une interconnexion entre deux modules XBee.

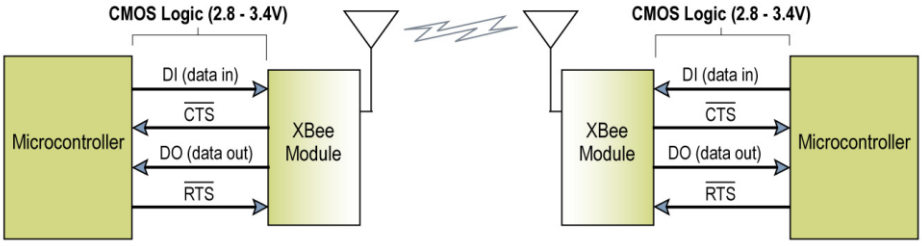


Figure 5.10 Communication entre deux modules XBee [Digi International, 2009]

5.3.2 RFID

L'utilisation d'un badge pour accéder à un local, enregistrer ses horaires de travail ou valider un titre de transport dans le bus ou le métro sont des gestes entrés dans le quotidien de bon nombre d'entre nous. Nous utilisons, sans en être toujours conscient, des technologies de

capture automatique de données basées sur les ondes et les rayonnements radiofréquence connues sous le nom de RFID.

La RFID se définit donc par technologie d'identification automatique qui utilise le rayonnement radiofréquence pour identifier les objets porteurs d'étiquettes lorsqu'ils passent à proximité d'un interrogateur. Ceci dit, la RFID ne peut pas se résumer à une seule technologie. En effet, il existe plusieurs fréquences radio utilisées par la RFID, plusieurs types d'étiquette ayant différents types de mode de communication et d'alimentation.

Pour transmettre des informations à l'interrogateur (plus généralement appelé lecteur), un tag RFID est généralement muni d'une puce électronique associée à une antenne (voir la figure 5.11). Cet ensemble est ensuite emballé pour résister aux conditions dans lesquelles il est amené à opérer. L'ensemble ainsi formé est appelé tag, label ou encore transpondeur.

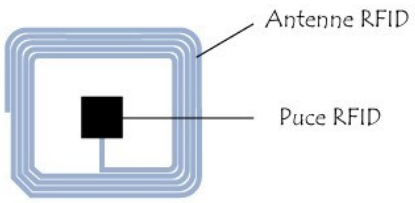


Figure 5.11 Tag RFID

L'identification électronique se divise en deux branches:

(1) L'identification «à contact»: Il s'agit de dispositifs comportant un circuit électronique dont l'alimentation et la communication sont assurées par des contacts électriques. Les deux principaux exemples d'identification à contact sont:
   a) Les circuits «mémoire»: ils comportent des fonctions mémoire embarqués sur des modules de formes et de tailles variées;
   b) Les cartes à puces: Les exemples de cartes à puces les plus connus sont les cartes bancaires et la carte SIM (*Subscriber Identity Module*).

(2) L'identification «sans contact»: On peut décomposer les identifications sans contacts en trois sous-branches principales:

a) La vision optique est un type de liaison qui nécessite une vision directe entre l'identifiant et le lecteur (laser). La mise en œuvre la plus répandue de cette technologie est le code à barre linéaire.
b) La liaison infrarouge assure quant à elle un grand débit d'information et une bonne distance de fonctionnement. Ces systèmes nécessitent également une visibilité directe.
c) Les liaisons radiofréquences permettent surtout la communication entre l'identifiant et un interrogateur, sans nécessité de visibilité directe. De plus, il est également possible de gérer la présence simultanée de plusieurs identifiants dans le champ d'action du lecteur (anticollisions).

Les tags RFID peuvent être classés en trois groupes:

(1) Le tag RFID passif qui rétro-module l'onde issue de l'interrogateur pour transmettre des informations. Il n'intègre pas d'émetteurs RF. Le tag passif utilise généralement l'onde (magnétique ou électromagnétique) issue de l'interrogateur pour alimenter le circuit électronique embarqué.
(2) Le tag RFID passif assisté par batterie comporte une alimentation embarquée (piles). Cette dernière n'est pas utilisée pour alimenter un émetteur puisque le principe de communication reste la rétro-modulation (comme pour le tag passif), mais pour alimenter le circuit électronique du tag ou tous autres circuits ou capteur connecté au circuit de base. Cette alimentation présente, en théorie, l'avantage d'améliorer les performances. Ce tag est largement utilisé pour des applications nécessitant une capture d'information (température, choc, lumière, etc.) indépendante de la présence d'un interrogateur.
(3) Le tag RFID actif embarque, quant à lui, un émetteur RF. La communication entre l'interrogateur et le tag est donc de type pair à pair. Ce tag embarque généralement une source d'énergie.

5.3.3 Cartes à puce RFID

Tel que mentionné précédemment, notre plateforme offre aussi un service de transport d'agents au moyen des cartes à puce sans contact. Celles-ci appartiennent à la famille de

produits *RFID*. Même si les différents RFID ont des utilisations qui peuvent être très différentes, tous reposent sur des principes communs que l'on retrouve dans les cartes à puce sans contact.

Comme le montre la Figure 5.12, un composant RFID et son lecteur ne nécessitent aucune liaison physique pour échanger les informations. En fait, un champ électromagnétique véhicule les données. L'alimentation du dispositif RFID ne comporte aucune source d'énergie interne. Le composant RFID doit extraire de ce champ électromagnétique l'énergie qui lui permet de fonctionner ainsi que les données qu'il doit recevoir du lecteur, mais il doit en outre être capable d'influencer ce champ pour envoyer à son tour des données au lecteur [Tavernier, 2007]. C'est cette dualité de rôle du champ électromagnétique qui fait toute la complexité des systèmes sans contact.

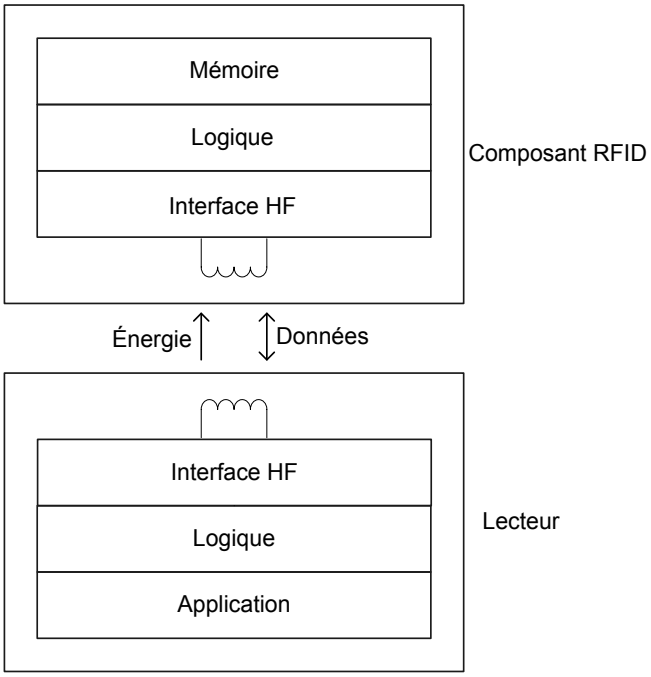


Figure 5.12 Principe générale d'un composant de type RFID [Tavernier, 2007]

5.3.4 Types de cartes à puce (avec ou sans contact)

Les cartes (à contact et sans contact) possèdent des caractéristiques communes, à savoir: leur dimension d'abord, la présence d'une mémoire permettant de stocker des données, l'intégration de la mémoire et éventuellement d'un processeur dans un seul circuit électronique, et enfin,

l'inscription dans un endroit réservé de la mémoire d'un numéro de série unique et permanent. Il existe deux principales familles de carte à puce distinctes car leurs modes d'utilisation sont complètement différentes.

Cartes à mémoire

La famille de cartes à mémoire est appelée aussi cartes synchrones en raison de son protocole de dialogue. Cette famille de carte est constituée de:

- Cartes à mémoire simple utilisés comme systèmes permettant de décrémenter des unités stockées au préalable (unités prépayées pour transport en commun, par exemple). Ces dispositifs emploient des mémoires inscriptibles, comportant des cellules élémentaires qui sont rendues inutilisables au fur et à mesure de la consommation. Elles ne peuvent évidemment pas être reprogrammées par l'utilisateur.
- Cartes à mémoire avec logique câblée qui comporte un dispositif de protection des données procurant un certain niveau de sécurité. Ces cartes peuvent également être chargées par des unités de communication.

Les cartes à mémoire avec logique câblée sont plus sûres que celles à mémoire simple mais elles ne permettent pas la mise en place des applications plus complexes. Pour ce faire, la carte doit avoir une «intelligence» locale que les cartes à mémoire simples ne disposent pas.

Cartes à microcontrôleur

L'intelligence locale qui fait défaut aux cartes à mémoire existe dans les cartes à puce à microcontrôleur, appelées aussi cartes asynchrones en raison de leur protocole de dialogue. Les cartes à puces «intelligentes» comprennent une unité centrale de microprocesseur, une mémoire ROM, une mémoire RAM, une mémoire EEPROM, une interface d'entrée/sortie série et toute la logique nécessaire pour faire fonctionner le tout.

Cartes à puce sans contact

Même si elles ne bouleversent pas la classification précédente, les cartes à puce sans contact semblent y semer une certaine confusion que nous tenons à expliquer. En effet, comme dans le cas précédent, il existe deux types des cartes à puce sans contact: Les cartes à mémoire sans contact et les cartes à microcontrôleur sans contact. Par contre, les cartes à puce sans contact dites à mémoire intègrent, entre autres, un microcontrôleur comme le montre la figure 5.13. Ce microcontrôleur est cependant peu utilisé: il ne sert que pour les fonctions de gestion de la partie applicative de la carte. Il sert essentiellement à la gestion du dialogue entre la carte et le lecteur, à l'anticollision, aux corrections automatiques des erreurs de dialogue éventuelles, etc. Son intervention au niveau applicatif se limite le plus souvent à la gestion des mots de passe autorisant ou non l'accès aux différentes zones de mémoire de la carte [Tavernier, 2007].

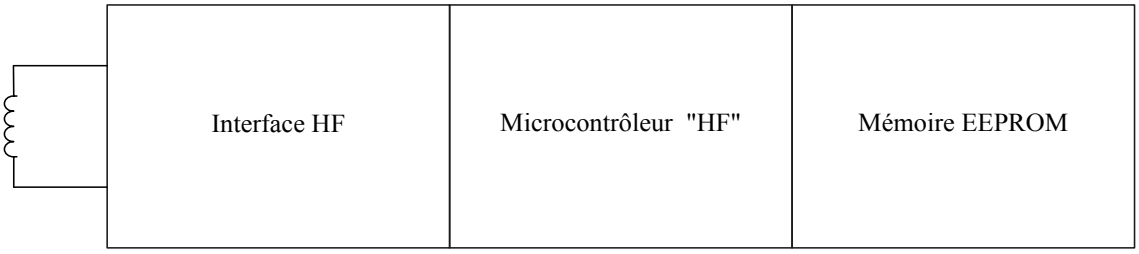


Figure 5.13 Modèle d'une carte à mémoire sans contact [Tavernier, 2007]

Dans une carte à microcontrôleur sans contact, on retrouve non seulement un microcontrôleur de gestion de toute la partie haute fréquence comme qu'on retrouve dans la carte à mémoire sans contact, mais aussi un autre microcontrôleur chargé des applications purement «carte» comme nous le montre la figure 5.14.

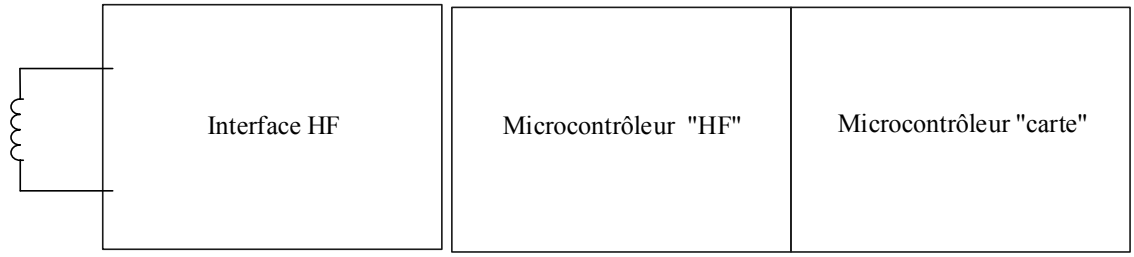


Figure 5.14 Modèle d'une carte à microcontrôleur sans contact [Tavernier, 2007]

Si l'on considère que, sur les figures 5.13 et 5.14, l'ensemble constitué par l'interface haute fréquence et le microcontrôleur de gestion haute fréquence remplace les contacts d'une carte à puce classique, on retrouve donc bien les deux familles principales que nous avons présentées.

Pour promouvoir l'interopérabilité entre les cartes à puce sans contact et leurs lecteurs, les quatre normes suivantes ont été proposées:

- ISO 14443-1 précise les caractéristiques physiques de la carte;
- ISO 14443-2 indique comment alimenter la carte à distance et comment décrire les signaux électriques pour établir le dialogue avec la carte;
- ISO 14443-3 définit les phases d'initialisation des échanges et de gestion de collision;
- ISO 14443-4 définit le protocole de transmission.

L'établissement du dialogue entre une carte à puce sans contact et son lecteur est un peu plus compliqué qu'avec une carte classique principalement à cause de la possible présence simultanée de plusieurs cartes au voisinage de ce dernier. Les procédures de l'établissement de dialogue entre une carte à puce sans contact et son lecteur se déroulent sommairement comme suit: lorsqu'une carte arrive dans la zone d'influence d'une antenne de lecteur et qu'elle se trouve donc suffisamment et correctement alimentée, elle effectue un *reset* interne et attend une commande provenant de ce dernier.

Le lecteur commence le dialogue en envoyant une commande appelée *Request* à laquelle la ou les cartes situées dans sa zone d'influence répondent par un *ATQA* (pour *Answer To Resquest Code*). Le processus d'anticollision entre en action et le lecteur lit l'identifiant transmis par la carte. Cet identifiant est en fait un numéro de série unique qui a été programmé dans la carte lors de sa fabrication.

Dans le cadre de ce projet, nous avons utilisé des cartes à mémoire sans contact (de *Mifare Classic* de 1 kilooctet et de 4 kilooctets de la compagnie NXP) [Mifare Classic, 2002] pour stocker le contexte d'exécution de l'agent qui représente l'utilisateur. En effet, dans le contexte de cette application, nous avons seulement besoin d'un espace mémoire qui permet la persistance de l'agent afin de le relancer dans un autre point d'accès. Cette persistance

pourrait se faire par d'autres supports comme des cartes SD ou autres. En effet, notre choix est surtout motivé par les commodités offertes par les cartes à mémoire sans contact.

Chaque carte *Mifare* possède un identifiant unique. En fonction des critères propres au processus d'anticollision utilisé, le lecteur choisit d'établir le dialogue avec une carte et il informe toutes celles qui se trouvent dans la zone d'influence [Tavernier, 2007]. Les cartes non sélectionnées retournent à un état semi-passif pendant lequel elles attendent une commande *Request* afin de pouvoir à nouveau «tenter leur chance». Puis, la carte sélectionnée envoie au lecteur une réponse de type *ATS* (pour *Answer To Select*) qui l'informe du type de carte avec laquelle il doit s'attendre à dialoguer. À partir de cet instant, les échanges utiles peuvent commencer.

5.4 Évaluation de la plateforme d'agents mobiles

Pour vérifier et valider le bon fonctionnement de notre plateforme d'agents mobiles, nous avons effectué un certain nombre des tests sur différents réseaux.

5.4.1 Agents mobiles sur un réseau filaire

Pour réaliser notre réseau filaire, nous avons monté un système composé de deux nœuds. Chaque nœud est composé d'un module ARM7 (*LPC-H2214* ou *LPC-H2294*) et un PC servant à la fois un serveur de code et une interface d'affichage des résultats.

Comme nous le montre la figure 5.15, la communication entre les PCs et les microcontrôleurs se fait via l'UART0 en utilisant le protocole de communication série. Pour faire communiquer entre les deux microcontrôleurs, l'UART1 du LPC-H2214 du nœud source est connecté directement à l'UART1 du LPC- LPC-H2294 du nœud destination.

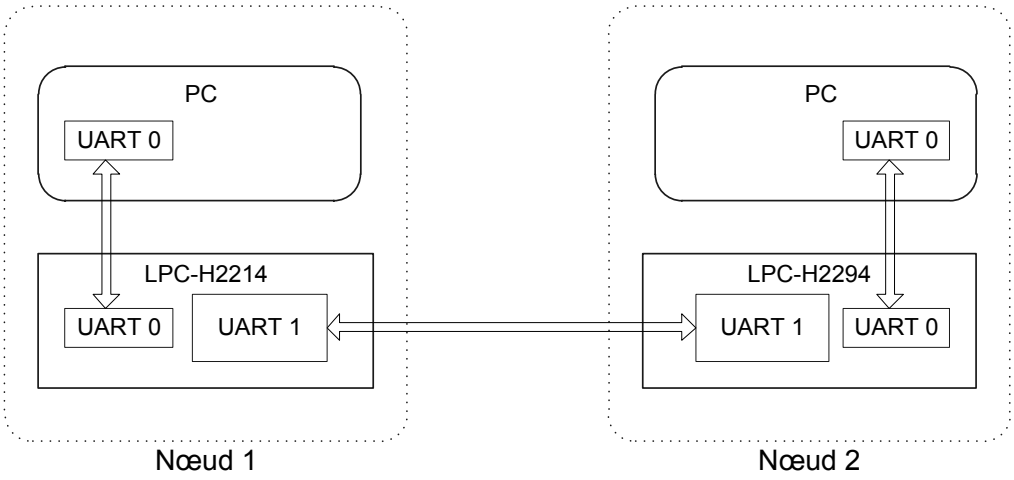


Figure 5.15 Modèle du réseau filaire

Pour tester notre plateforme sur ce réseau filaire, nous avons réalisé un exemple d'application d'un agent mobile qui réalise les étapes suivantes:

1. L'agent initialise un certain nombre de variable de différents types (*char*, *short*, *int*, *float*, *double*) dans le nœud 1;
2. Il effectue des opérations sur ces variables et affiche leur contenu, puis il se déplace vers le nœud 2;
3. L'agent poursuit ses opérations, affiche le contenu des variables et retourne au nœud de départ, le nœud 1;

Dans cette expérience, l'agent effectue différentes opérations. À titre d'exemple, l'agent effectue des opérations arithmétiques telle que l'addition, la soustraction, etc. Il effectue également des concaténations des chaînes de caractères contenues dans des variables provenant d'un nœud distant.

Cette expérience nous a permis de valider la mobilité forte qui consiste à interrompre l'exécution de l'agent sur le nœud source, ensuite le code est transféré du serveur vers le nœud destination. Puis, les données représentant l'état de l'agent sont transférées du nœud source vers le nœud destination. Enfin, lorsque l'agent est arrivé au nœud destination, il poursuit son exécution là où elle avait été interrompue sur le nœud de départ. Le fil d'exécution de cette expérience est visualisé sur l'écran du PC qui est connecté sur le microcontrôleur de chaque nœud.

Notez que notre système de migration d'agents utilise seulement la période du *temps mort* qui est égal à la durée de la migration totale. Avec un exemple d'application d'un agent mobile de 528 kilooctets, nous avons mesuré un *temps mort* de 1 minute et 35 secondes. Ce temps dépend essentiellement la vitesse de transmission du lien de communication et la vitesse d'effacement et d'écriture des données dans la mémoire Flash du nœud d'accueil. Il est pertinent de préciser que la migration d'un code vers un nœud de destination est équivalente à la programmation de la mémoire flash de ce nœud.

5.4.2 Agents mobiles sur un réseau Zigbee

Pour réaliser notre réseau Zigbee, nous avons monté un système composé de six nœuds. Chaque nœud du réseau constitue:

1. Un module ARM7 LPC-H2214/LPC-H2294;
2. Un module XBee série 2;
3. Un ordinateur de bureau (PC) pour suivre le fil d'exécution.

La figure 5.16 présente Module XBee série 2 [Digi International, 2009]. Ce module est un petit modem radio qui fonctionne sur le principe des normes 802.15.4 et/ou Zigbee. Il utilise une liaison série sans-fil pour communiquer avec les autres modules. Le module est constitué de:

- 20 broches au format DIL (pour *dual in-line package*);
- antenne planaire.

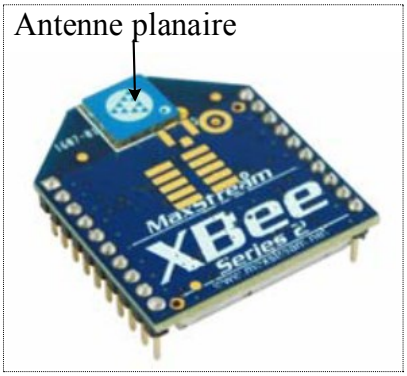


Figure 5.16 Module XBee série 2

Pour tester notre plateforme sur le réseau Zigbee, nous avons réalisé un exemple d'application d'agents mobiles similaire à celui du réseau filaire. La différence entre les deux expériences se situe au niveau du service de transport, du nombre de nœuds utilisés ainsi que du nombre de tours effectués par l'agent. Comme nous le montre la figure 5.17, nous avons utilisé six nœuds dans cette expérience. Un tour est le parcours de l'agent des nœuds: 1-2-3-4-5-6-1.

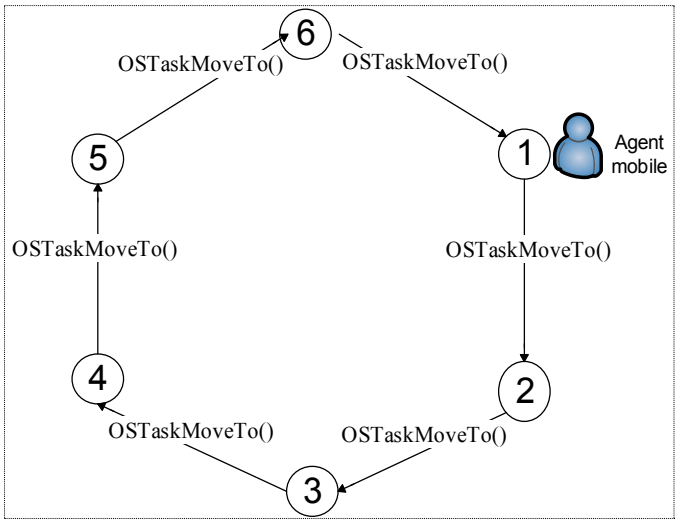


Figure 5.17 Itinéraire de l'agent mobile dans le réseau Zigbee

De plus, contrairement à l'expérience précédente où l'agent n'effectue qu'une seule itération, dans celle-ci l'agent fait cinq fois le tour de nœuds. L'agent débute son itinéraire sur le nœud 1. Comme l'expérience précédente, celle-ci nous aussi a permis de valider la mobilité forte supportée par notre plateforme d'agents mobiles.

5.4.3 Agents mobiles sur des cartes à puce RFID

Nous allons décrire de façon détaillée un exemple d'application exploitant notre solution de la mobilité d'agents sur des cartes à puce RFID (le système de navigation pédestre) dans le chapitre prochain. Mais dans cette section, nous allons présenter le résultat d'une expérience permettant de mesurer le *temps de transmission* de l'état d'exécution d'un agent mobile. Cette expérience consiste à capturer l'état d'un agent mobile s'exécutant sur un nœud source, puis à le transférer vers une carte à puce RFID. Ensuite, l'état de l'agent mobile est restauré dans un nœud de destination.

Afin d'évaluer les performances de la plateforme d'agents mobiles sur les différents réseaux, nous allons d'abord calculer le débit réel du port série (UART) de microcontrôleurs utilisés, en occurrence le LPC-H2214 et le LPC-H2294. Le protocole de communication série fonctionne comme suit. Lorsqu'un nœud source commence à communiquer, il prévient le nœud destination par un premier bit à zéro: c'est le bit de départ. Puis, il envoie les 8 bits de données, suivi d'un bit d'arrêt pour indiquer au nœud destination que la trame est terminée. Pour un débit de transmission de 115 200 bps et une fréquence du processeur de 60 Mhz, le calcul de la valeur du diviseur *DIV* se fait comme suit:

$DIV = \frac{\text{fréquence du processeur}}{16 \text{ x } 115\ 200} = \frac{60 \text{ x } 10^6}{16 \text{ x } 115\ 000} = 32.552.$ Ceci va déterminer le débit de la transmission du lien de communication.

Ce nombre peut être approximée à 32. Ceci sert à programmer les paramètres *DLL* (pour *Divisor Latch LSB Register*) et *DLM* (pour *Divisor Latch MSB Register*) de l'UART. Ce nombre doit être converti en hexadécimal avant d'être écrit dans *DLL* et *DLM*. La valeur hexadécimale de DIV est 0x20. Par conséquent, 0x20 doit être écrit dans *DLL* et 0x00 dans *DLM*. Le débit réel de transmission (*drt*) de l'UART est:

$drt = \frac{60 \text{ x } 10^6}{32 \text{ x } 16} = 117\ 187.5 \text{ bps}.$

Il en résulte une période:

$T = \frac{1}{drt} = \frac{1}{117\ 187.5} = 8.53\ \mu s.$

Le temps de transmission de la trame (*ttrm*) peut être calculé comme suit:

$ttrm = (trm) \text{ x } (T) = (10 \text{ bits }) \text{ x } (8.53\ \mu s) = 85.3\ \mu s.$

En sachant la taille du contexte d'exécution (*tce*), nous pouvons calculer son temps de transfert dans le réseau filaire (*ttrf*) comme suit:

$ttrf = (\text{tce}) \text{ x } (ttrm) = (34309 \text{ octets}) \text{ x } (85.3\ \mu s) = 2.926 \text{ s}.$

Ce temps de transfert du contexte d'exécution ne tient pas compte des éventuelles pertes de trames et les délais pour les retransmissions. Le tableau 5.1 résume l'évaluation quantitative des trois réseaux expérimentés dans ce projet. Comme nous le présente le tableau 6.1, le

temps de réponse du point d'accès est environ 10 secondes. Ceci est le temps de lecture et d'écriture de la carte à puce sans contact par le point d'accès. En effet, le point d'accès affiche le chemin à prendre une fois que l'agent retourne vers la carte à puce sans contact.

Tableau 5.1 Évaluation de performances de la plateforme proposée avec différents services de transport

	Réseau filaire	Réseau Zigbee	Lecteur/encodeur RFID
Taille de la pile utilisée (octets)	1024	1024	768
Taille de l'état d'exécution mise en format de transfert (octets)	3011	3011	2755
Débit de transmission (bps)	115200	115200	11520
Temps de transmission de l'état d'exécution au format de transfert (secondes)	1.41	2.88	4.52

Nous constatons que les trois réseaux présentent différents temps de transmission de l'état d'exécution de l'agent même s'ils ont le même débit de lien de communication et quasiment la même taille de données à transférer (voir le tableau 5.1). En effet, le réseau filaire présente le temps de transmission de l'état d'exécution le plus court (1.41 secondes). Le réseau de lecteur/encodeur RFID présente le temps de transmission le plus lent (4.52 secondes). Ceci est dû essentiellement par les pertes de trames et les délais pour les retransmissions.

5.5 Conclusion

Dans ce chapitre, nous avons présenté la mise en œuvre de la plateforme d'agents mobiles pour systèmes embarqués µC/MAS. En effet, comme cette plateforme est mise en œuvre sous forme d'une bibliothèque de programmes, nous avons exposé les primitives essentielles qui la constituent. Cette bibliothèque est intégrée aux fonctionnalités du noyau temps réel µC/OS-II. La bibliothèque de la plateforme d'agents mobiles est construite selon une architecture modulaire qui permet son intégration avec d'autres services en utilisant le noyau temps réel µC/OS-II et différents moyens de transport pour les agents. Elle comporte les modules suivants à la base de son intégrabilité avec ces services:

- Un module de capture/restauration qui assure l'interface avec le noyau temps réel;
- Un module d'encodage/décodage qui permet une mise en forme de données courantes;
- Un module d'encodage/décodage qui permet une mise en forme du contexte de l'agent;

- Un module qui assure l'interface entre différents services de transport. Ces derniers peuvent être synchrones (comme dans le cas d'une liaison série, d'un réseau ZigBee) ou asynchrones (comme dans le cas de l'utilisation d'une carte intelligente de stockage RFID).

Pour récapituler, le mécanisme de migration des agents fonctionne sommairement comme suit: Dans un premier temps, le code de l'agent est chargé depuis un serveur. Dans un deuxième temps, les données courantes de l'agent sont envoyées vers le nœud de destination et elles sont reconstruites dans le même type de blocs de partition que sur le nœud source. Dans un troisième temps, l'agent sur le nœud source est suspendu et son contexte est envoyé à la destination. Enfin, ce contexte est utilisé pour créer une nouvelle tâche qui va héberger la suite du fil d'exécution de cet agent sur le nœud de destination.

Nous avons utilisé deux primitives (*OSMemBlckMove()* et *OSTaskMoveTo()*) pour la migration afin de séparer le contexte d'exécution et les données courantes pour de raison de flexibilité et d'optimisation. En effet, c'est le développeur de l'application qui décide les données courantes à transférer d'un nœud à l'autre en utilisant la directive de migration que nous avons spécifiée.

Pour évaluer notre plateforme d'agents mobiles, nous avons développé une application de test sur un réseau filaire, un réseau Zigbee et sur des cartes à puce RFID. Dans les trois réseaux expérimentés, la plateforme d'agents mobiles natifs pour systèmes embarqués répondent aux spécifications prédéfinies. La bibliothèque qui met en œuvre la plateforme d'agents mobiles pour systèmes embarqués compte 7854 lignes de code. Celle-ci n'inclut pas le code du noyau temps réel. En revanche, elle inclut un certain nombre de lignes de code pour le déverminage. Notons que la plateforme est un état de prototype, donc un certain nombre d'optimisation peut être apportée. La mise en œuvre de la plateforme sous forme d'une bibliothèque permet la modularité, la réutilisation et ainsi la réduction de la taille de code.

CHAPITRE 6
EXEMPLE D'APPLICATION EXPLOITANT NOTRE SOLUTION DE LA MOBILITÉ D'AGENTS ET ÉVALUATION

Dans le chapitre précédent, nous avons exposé la mise en œuvre de la solution de mobilité d'agents proposée. Dans ce chapitre, nous présentons un exemple d'application exploitant la solution de la mobilité d'agents, un système de navigation pédestre. Pour commencer, nous passons en revue les systèmes de navigation pédestre existants. Ensuite, nous décrivons notre système de navigation pédestre. Enfin, nous finissons par une comparaison de notre système de navigation avec des systèmes similaires.

6.1 Introduction

Imaginons un centre commercial, un aéroport, un hôpital, un musée, etc. dans lesquels les panneaux indicateurs, les écrans, les dispositifs de communication se mettent instantanément à notre service dès que nous en franchissons le seuil. Ainsi, nous entrons dans l'ère de l'informatique omniprésente et sortons dans celle de l'ordinateur personnel. En effet, des multitudes de composants électroniques s'insèrent dans l'environnement et dans les objets du quotidien. Ces composants électroniques sont capables de se repérer dans l'espace, de se reconnaître les uns les autres et de se relier en réseau sans fil. Ainsi, chaque individu se déplace entouré de sa «bulle de communication» et, selon l'endroit où il se trouve, interagit avec les bulles d'autres individus ou des objets situés dans son environnement. C'est dans cette nouvelle ère de l'informatique omniprésente que notre solution de mobilité d'agents apporte pleinement sa contribution. Rappelons que l'informatique omniprésente est généralement mise en œuvre par des objets communicants (systèmes embarqués) avec des contraintes fortes en termes de mémoire, de consommation d'énergies, de bande passante, etc.

Les applications envisagées sont sans limites, si ce ne sont celles de notre propre imagination. Elles peuvent aller de réseaux à quelques dizaines de nœuds pour des scénarios domotiques à des millions de nœuds pour gérer une multitude de services à l'échelle d'une grande ville. Dans la domotique (appelée aussi l'habitat intelligent), on peut imaginer une application où un agent mobile suit le déplacement d'un utilisateur. L'agent doit migrer d'un nœud à autre pour être toujours au même endroit que l'utilisateur afin de lui apporter une assistance dans ses tâches quotidiennes. En effet, une tâche quotidienne de l'utilisateur peut être, par exemple, de prendre des médicaments à une heure précise de la journée. Si l'utilisateur oublie de prendre les médicaments, l'agent va, dans ce cas, lui rappeler. À l'aide des capteurs, un agent intelligent mobile peut identifier l'activité en cours à un endroit spécifique de l'habitat (comme la cuisine, par exemple) et ainsi apporter une assistance adéquate à l'utilisateur.

Un autre exemple d'application où notre solution de la mobilité d'agents peut apporter sa contribution est la collecte d'informations en temps réel dans des sites difficiles d'accès comme le forage du pétrole. Dans cette perspective, on peut concevoir une grappe où chaque nœud est constitué d'un microcontrôleur et d'un lecteur de tags RFID (voir la figure 6.1). À titre d'exemple, la compagnie *IDENTEC SOLUTIONS* conçoit des lecteurs (*i-Port M 350-2 Reader*) de tags RFID actives (*i-Q350*) permettant d'opérer à une distance allant jusqu'à 250 mètres. Dans ce cas, un agent mobile permettrait de se déplacer d'un nœud à un autre afin de collecter en temps réel des informations:

- D'identification;
- De localisation;
- De surveillance et de traçabilité d'individus ou d'objets mobiles;
- De prises de mesures (comme la pression, le débit, le niveau, la température, etc.) disposées sur des équipements.

Un autre exemple d'application où notre solution de la mobilité d'agents peut apporter sa contribution est le domaine de la navigation pédestre qui est l'objet de ce chapitre. Ce domaine prometteur s'inscrit dans un contexte global de mobilité et particulièrement dans les espaces urbains ou à l'intérieur des bâtiments. Dans ce chapitre, nous allons présenter un prototype de navigation pédestre exploitant notre solution de mobilité d'agents. Toutefois,

avant de présenter notre prototype, nous allons d'abord passer en revue les systèmes de navigation pédestre existants.

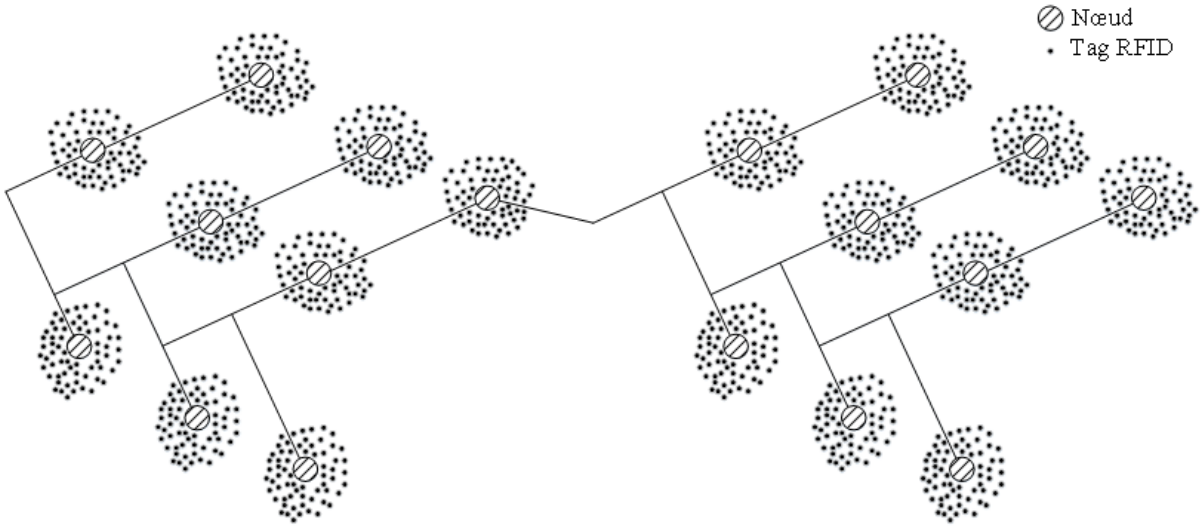


Figure 6.1 Grappe de nœuds

6.2 Systèmes de navigation pédestre existants

Depuis la carte et la boussole jusqu'à l'avènement du *GPS (Global Positioning System)*, l'Homme a développé des moyens techniques pour faciliter ses déplacements et augmenter son autonomie. Un système d'assistance à la navigation sous-tend de se localiser en continu, de planifier sa trajectoire puis d'être guidé en tenant compte des obstacles éventuels. Pour un véhicule, le GPS doit être associé à une carte numérique. Le GPS est le système de référence pour localiser les véhicules, les navires ou les avions. Cependant, il ne permet pas la localisation pédestre car sa précision est insuffisante et il est souvent inopérant dans les espaces urbains ou à l'intérieur des bâtiments. D'autres technologies s'appuyant sur des réseaux de balises communicantes (Wi-Fi, Zigbee, RFID, etc.) sont en cours d'expérimentation, mais à ce jour, aucune n'est encore opérationnelle et la localisation pédestre reste un défi [Elloumi, 2012] [Elloumi et al., 2011].

Contrairement à la localisation de véhicules, la localisation pédestre doit faire face à de nombreux problèmes. Premièrement, la précision de localisation doit être accrue pour atteindre de l'ordre de quelques dizaines de centimètres à un mètre. La localisation des

personnes doit être opérationnelle aussi bien à l'extérieur qu'à l'intérieur des bâtiments pour assurer des déplacements urbains complets.

Deuxièmement, dans le cas des véhicules, la navigation s'appuie généralement sur un modèle de mouvement parfaitement maîtrisé. Dans le cas d'un piéton, le système de navigation doit avoir une plus grande réactivité pour faire face aux mouvements plus rapides et moins prévisibles du piéton que ceux d'un robot ou d'un véhicule qui sont facilement prévisibles. En effet, il est plus difficile de modéliser le déplacement d'un piéton qu'une roue car le pas humain peut varier d'une personne à une autre et même pour le même individu selon son état courant et selon les circonstances. Cette variation peut devenir une source d'erreur non négligeable puisque sur des milliers de pas, l'incertitude sur la distance parcourue croît rapidement.

Les principales technologies utilisées pour la localisation des personnes s'appuient sur des systèmes communicants ou des capteurs déjà utilisés dans la navigation automobile. Celles-ci peuvent être classées en deux catégories. La première met en œuvre un bouquet d'émetteurs ou balises et un récepteur (GPS, UWB, Wi-Fi, Bluetooth, RFID, etc.) tandis que la seconde s'appuie sur un ou plusieurs capteurs autonomes (MEMS, caméra, etc.) sans dépendre d'une infrastructure [Elloumi, 2012].

6.2.1 Systèmes de localisation dépendants d'une infrastructure

Dans cette première famille de systèmes de localisation, on utilise des réseaux de capteurs comme les réseaux satellitaires (GPS), des balises radiofréquence (RFID) ou encore les réseaux locaux existants (Wi-Fi, GSM) [Elloumi, 2012]. Ce type de système de localisation demande l'installation d'une infrastructure bien déterminée dans toutes les régions où on veut assurer la localisation avec une précision suffisante. Nous pouvons citer l'exemple du système de navigation pédestre proposé par une équipe de chercheurs [Ran et al., 2004] de l'Université de Floride à Gainesville pour guider les déficients visuels au cours de leurs déplacements au cœur du campus (à l'extérieur ou bien à l'intérieur de ses bâtiments). Ce système de navigation pédestre est équipé d'un dispositif qui associe le GPS différentiel (DGPS) à un

système d'information géographique (SIG) pour la localisation à l'extérieur [Helal et Moore, 2001] et utilise un système de positionnement ultrason pour la localisation à l'intérieur. Le récepteur est composé de 2 balises attachées aux épaules de l'utilisateur alors que les émetteurs sont constitués par 4 pilotes ultrasons montés dans les quatre coins du bâtiment pour fournir les mesures de la localisation de la personne. Dans différentes localisations, les résultats montrent une précision de 22 cm sur des trajets intérieurs et extérieurs [Ran et al., 2004].

Dans [Kulyukin et al., 2004a] [Kulyukin et al., 2004b] [Kulyukin et al., 2004c], les auteurs ont proposé une approche de navigation à l'intérieur des bâtiments pour personnes avec une déficience visuelle. Cette approche utilise des RFID pour créer un réseau de communication pour permettre la localisation pédestre. Dans cette approche, les auteurs se sont inspirés du concept du chien-guide pour le développement de leur système de navigation pédestre. Ce système est composé d'une plateforme robotique, d'un dispositif de navigation, d'un récepteur RFID et des radio-étiquettes pour la localisation. La plateforme robotique est rattachée au bout d'une laisse comme un substitut à un chien-guide. Cette plateforme possède 3 roues et 16 sonars ultrasons, 8 en avant et 8 en arrière. Le dispositif de navigation comprend un ordinateur portable qui est connecté au microcontrôleur du robot via un câble USB afin de le guider. L'ordinateur portable est connecté aussi à un récepteur RFID pour assurer la localisation du robot guide. Ce prototype a été testé sur 5 déficients visuels, dont 3 sont complètement aveugles et 2 qui pouvaient seulement percevoir la lumière sur un trajet intérieur de 40 m de deux bâtiments inconnus pour eux. Tous les participants sont parvenus à atteindre leurs destinations mais ils se sont plaints de la vitesse lente du robot guide qui est 0.5 m/s ainsi que de ses mouvements saccadés [Kulyukin et al., 2004a]. La vitesse de marche normale se situe entre 1.2 et 1.5 m/s.

Dans [Ertan et al., 1998], les auteurs ont proposé un système de localisation à infrarouge. Ce système est composé de trois unités principales: un gilet qui contient une grille de 4x4 micromoteurs pour délivrer des signaux de guidage haptiques sur le dos de l'utilisateur, un ordinateur portable pour la planification d'itinéraire et un récepteur et des émetteurs à infrarouge pour localiser la personne. Les résultats de tests ont montré de 0 à 3 erreurs par

trajet intérieur de 15 à 21.3 m [Ertan et al., 1998]. Nous constatons que ce système est moins précis que les deux premiers.

6.2.2 Systèmes de localisation autonomes

À la différence de la première famille, ce type de systèmes de localisation ne nécessite aucune infrastructure existante [Elloumi, 2012]. Il repose généralement sur un système de navigation pédestre (SNP) porté par la personne. Ce SNP peut être couplé avec une base de données cartographiques des régions ou bien des bâtiments. Dans [Gilliéron et al., 2004], les auteurs s'inspirent de la théorie des graphes et créent un modèle lien/nœud pour la construction de la carte du bâtiment. La position de la personne est calculée à l'aide d'un module de navigation pédestre développé au sein du laboratoire de l'EPFL (École polytechnique fédérale de Lausanne) [Ladetto et Merminod, 2002]. Le SNP est constitué d'un récepteur GPS, d'un compas magnétique numérique, d'un gyroscope, d'un baromètre et des algorithmes embarqués *DR (Dead Reckoning)* [Elloumi, 2012] [Ladetto et Merminod, 2002].

Dans [Hesch et Roumeliotis, 2007], les auteurs ont proposé un système de localisation (à l'intérieur de bâtiments) pour personnes présentant une déficience visuelle. Ce système comporte un odomètre monté au pied de l'utilisateur pour mesurer sa vitesse et une canne blanche sur laquelle sont attachés deux capteurs: un gyroscope triaxial et un scanner laser pour estimer précisément l'attitude de la canne. Les informations provenant de l'odomètre et de deux capteurs sont fusionnées en deux étapes pour l'estimation de la pose de l'utilisateur (son orientation et sa position) [Elloumi, 2012]. La première étape utilise les mesures inertielles du gyroscope triaxial et les mesures de l'orientation relative du *scanner* laser afin d'estimer avec précision l'attitude de la canne blanche. La deuxième étape estime la position de la personne détenant la canne, en intégrant les mesures de la vitesse linéaire de l'odomètre, une version filtrée de l'estimation du mouvement de la canne et les primitives (des coins) extraites par le *scanner* laser. Ce système permet une précision de 16 cm sur un trajet intérieur de 130 m [Hesch et Roumeliotis, 2007].

Dans [Renaudin et al., 2007], les auteurs ont proposé une solution de navigation pédestre destinée pour les interventions d'urgence. Cette solution consiste en une conjonction des capteurs MEMS et des balises d'identification à fréquences radio RFID afin d'augmenter la précision et la robustesse du système. Ce système permet une précision de 5 m à l'intérieur d'un bâtiment de 225 m x 125 m [Renaudin et al., 2007]. Dans [Walder et al., 2009], les auteurs ont proposé un système de localisation pour assister et améliorer les interventions d'urgence à l'intérieur des bâtiments. Leur système de localisation autonome combine une centrale inertielle et un plan du bâtiment pour la localisation. Une centrale inertielle est un ensemble de capteurs permettant de mesurer le mouvement. En outre, ce système de localisation utilise un réseau local sans fil pour la communication entre ses différentes composantes. Ce système permet une précision inférieure à 2 m [Walder et al., 2009].

6.2.3 Systèmes de navigation pour personnes ayant une déficience cognitive

Les capacités spatiales des personnes dépendent principalement des quatre ressources suivantes: des capacités perceptrices, des capacités de traitement d'informations, des connaissances précédemment acquises et des capacités motrices [Chang et al., 2010]. Ces capacités forment une condition nécessaire pour que les personnes trouvent leur chemin qui mène à leur destination. Cependant, pour les personnes avec des déficiences cognitives sévères, les trois premières ressources sont généralement limitées.

Dans [Liu et al., 2006], les auteurs ont développé une infrastructure d'un système de navigation pédestre sans toutefois mettre en œuvre toutes les parties. En effet, l'utilisateur se déplace en suivant les instructions d'un dispositif à main pour se rendre à sa destination. Les instructions pour guider l'utilisateur sont une combinaison des messages texte, audio et image préenregistrés. Le dispositif à main affiche ces instructions à chaque point de décision (voir la figure 6.2). Le dispositif à main reçoit ces instructions depuis un serveur distant via la communication Wi-Fi. La question qu'on pourrait se poser est: comment le serveur distant détermine-t-il la position et l'orientation de l'utilisateur afin d'envoyer l'instruction appropriée? En effet, une personne, qui suit l'utilisateur, transmet en temps réel la position et l'orientation via un dispositif portable (une tablette) au serveur distant. Pour fournir les

informations de localisation de l'utilisateur au serveur distant, la personne utilise une interface utilisateurs graphique qui implémente la carte du site.

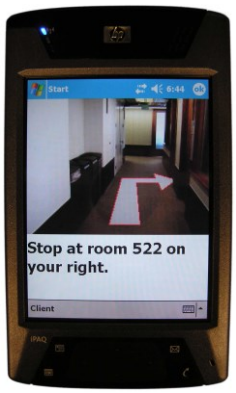


Figure 6.2 Exemple d'une instruction [Liu et al., 2006]

L'image du côté gauche de la figure 6.3 illustre une capture d'écran de l'application s'exécutant sur le serveur. Les marques vertes sur la carte du site illustrent les points où les images ont été prises. Ces images sont envoyées au dispositif à main pour guider l'utilisateur vers sa destination. La marque rouge solide montre les informations de localisation de l'utilisateur. L'image du côté droit de la figure 6.3 illustre une capture d'écran de l'application qui s'exécute sur la tablette de la personne suivant l'utilisateur. La marque rouge représente la localisation et l'orientation de l'utilisateur.

Dans [Chang et al., 2010] [Chang et al., 2009] [Chang et al., 2008], les auteurs ont proposé différents prototypes de navigation pédestre pour personnes ayant une déficience cognitive. Leur système comprend essentiellement un dispositif à main (*PDA, personal digital assistant*) servant comme interface utilisateur, des lecteurs et des tags RFID et un logiciel intégré mettant en œuvre un algorithme de routage. Contrairement au système de Liu [Liu et al., 2006], le travail proposé par Chang [Chang et al., 2010] détermine la localisation de l'utilisateur de façon automatique. En effet, une étiquette RFID est placée à chaque point de décision qui est une position physique lorsque plusieurs choix de navigation se présentent à l'utilisateur. Les points de décision où les choix de navigation doivent être faits peuvent être des entrées, des coins, des intersections de couloirs, etc.

Figure 6.3 Côté gauche illustre l'application s'exécutant sur le serveur et le côté droite présente celle sur la tablette de la personne suivant l'utilisateur [Liu et al., 2006]

Dans beaucoup de situations comme des couloirs droits dans des environnements intérieurs, aucune étiquette RFID ne doit être placée entre les points de décision car il n'y a pas de changements de directions. Par conséquent, les étiquettes RFID ne sont pas densément distribuées partout. Ceci réduit le nombre total des étiquettes RFID utilisées. Les utilisateurs doivent visuellement identifier et localiser une étiquette qui est habituellement sur le mur avant que leur PDA avec un lecteur RFID intégré ne puisse interagir avec l'étiquette dans une distance courte, généralement inférieure à 5 centimètres. Les étiquettes RFID sont associées aux LED clignotantes bleues pour augmenter leur visibilité.

6.3 Système de navigation pédestre proposé

Bien que les systèmes actuels répondent aux besoins de ses utilisateurs dans la plupart des cas, il existe cependant certaines situations où ceux-ci présentent des limitations. En effet, les systèmes pédestres pour personnes ayant une déficience cognitive utilisent exclusivement des dispositifs à main. Il est donc encombrant et confondant de marcher et en même temps suivre les instructions d'un dispositif à main pour certaines personnes qui présentent certaines déficiences. Par exemple, dans [Liu et al., 2006], les auteurs mentionnent qu'une personne en fauteuil roulant est incapable de tenir un dispositif à main. Le fait que l'utilisateur puisse être distrait en se concentrant trop sur l'appareil de navigation est aussi un danger potentiel car elle risque d'oublier de regarder sa route et entrer en collision avec des piétons ou des objets.

Pour remédier aux problèmes que nous venons de mentionner au paragraphe précédent, nous proposons un système de navigation pédestre où l'utilisateur n'est pas obligé de porter un dispositif à main. En effet, les dispositifs sont déjà disséminés dans un espace intelligent. Ce dernier consiste à un espace (comme un centre commercial) équipé de plusieurs nœuds (de petits dispositifs) où un agent mobile suit les déplacements de l'utilisateur d'un nœud à un autre pour le guider vers sa destination. Comme notre système utilise des dispositifs disséminés dans un espace intelligent, il est dans le type de localisation dépendant d'une infrastructure.

Le système à base d'agents mobiles communique avec l'utilisateur de façon personnalisée à l'aide d'une interface visuelle interactive. L'agent proposé dans ce projet est un programme informatique. Sa principale caractéristique est sa capacité de se déplacer de manière autonome entre plusieurs nœuds. Selon le principe d'autonomie, l'agent lui-même décide quand et où migrer. Ainsi, pour plusieurs applications, le paradigme d'agents mobiles offre l'autonomie, améliore la flexibilité, économise la largeur de bande passante et évite la latence de réseau.

Le système à base d'agents mobiles est conçu pour n'importe quel individu ayant des problèmes de mémoire et d'orientation spatiale et qui voyage dans des environnements intérieurs ou extérieurs, comme des centres commerciaux, des aéroports, des hôpitaux, des musées, etc. Ce système peut également être utile à toute personne qui se déplace dans un lieu inconnu. À notre connaissance, nous sommes les premiers à proposer un système à base d'agents mobiles opérant dans un espace intelligent pour guider des personnes.

En plus des exemples que nous avons présentés dans la section précédente, un autre exemple concerne les personnes qui ont besoin d'un écran qui affiche en gros caractères avec un contraste élevé. Une augmentation du contraste de l'écran entraîne une plus grande consommation d'énergie. Dans le même ordre d'idée, si la taille des caractères augmente, la taille de l'écran augmente ainsi que la consommation d'énergie du dispositif. L'utilisation de ce type d'écran crée deux problèmes:

1) la consommation d'énergie qui réduit l'autonomie de l'appareil,
2) la taille de l'écran qui rend la portabilité un défi pour certains utilisateurs.

En raison de la disponibilité de l'énergie électrique, nous proposons un grand nombre d'appareils fixes disséminés dans un espace. Chaque dispositif est équipé d'un écran relativement grand avec un contraste élevé. Ces dispositifs doivent détecter la présence de l'utilisateur et le guider vers sa destination. Ainsi, l'utilisateur peut visualiser le chemin nécessaire pour atteindre sa destination. Chaque point d'accès peut servir un grand nombre d'utilisateurs, mais seulement un à la fois. En effet, le nombre d'utilisateurs que le point d'accès doit servir ne présente aucune limite. L'interaction entre l'utilisateur et le point d'accès doit être courte et rapide afin de guider un grand nombre d'utilisateurs. En conséquence, il ne devrait y avoir aucune saisie à l'exception des points d'accès qui se trouvent aux entrées de l'espace de navigation. En effet, les points d'accès qui se trouvent aux entrées de l'espace de navigation doivent être équipés d'une interface utilisateur comme un écran tactile ou un clavier. Cette interface permet aux individus de construire leur profil personnel dont les données recueillies se transforment en agent mobile.

6.3.1 Approche proposée

Pour la construction du système de navigation pédestre, on pourrait envisager un certain nombre d'architectures possibles. Une première approche possible est l'utilisation d'un modèle client-serveur. Pour ce faire, chaque point d'accès agit comme un client connecté à un serveur distant. Dans cette approche, les données associées à chaque utilisateur sont stockés dans le serveur. Les clients et le serveur utilisent un protocole de passage des messages pour communiquer.

Avant que le point d'accès n'indique le chemin pour arriver à sa destination, l'utilisateur est identifié au moyen de la technologie RFID. Chaque fois qu'un utilisateur consulte le chemin pour arriver à sa destination sur un point d'accès, un nombre significatif d'échanges des messages a lieu entre le serveur et le point d'accès. Lorsque ce nombre est multiplié par un grand nombre de points d'accès, le trafic réseau augmente de façon significative. Ceci peut causer de la congestion du réseau et peut entraîner des retards, des interruptions et/ou la perte d'informations. La figure 6.4 montre le modèle client-serveur en combinaison avec la technologie RFID.

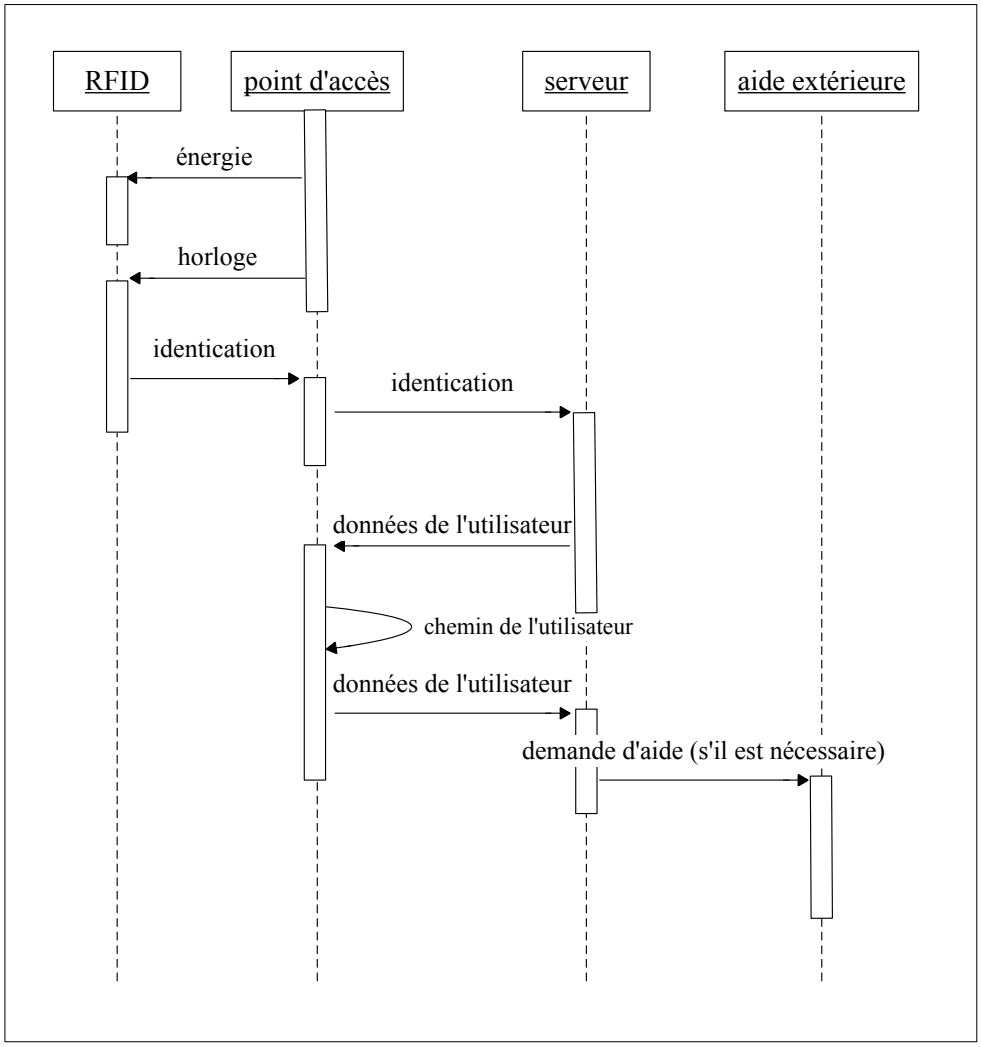


Figure 6.4 Client-serveur en combinaison avec la technologie RFID

Une deuxième approche possible est l'utilisation d'un paradigme d'agents mobiles. Avec cette approche, l'agent mobile suit l'utilisateur d'un point d'accès à l'autre pour le guider. Pour ce faire, l'agent (code + état) associé à chaque utilisateur est stocké dans un serveur distant. Lorsque l'utilisateur arrive à proximité d'un point d'accès, il est identifié par une étiquette RFID et le point d'accès demande le transfert de l'agent du serveur. Ensuite, l'agent est transféré au point d'accès courant. Comme l'approche précédente, il y aura beaucoup de trafic réseau lorsqu'un grand nombre de points d'accès demande le transfert de l'agent en même temps. Cela peut entraîner des retards, des interruptions et/ou une perte d'informations.

Une troisième approche possible est l'utilisation du paradigme d'agents mobiles en combinaison avec une carte à puce sans contact comme le montre la figure 6.5. Une carte à

puce est un petit ordinateur dans le format d'une carte de crédit sans interface homme-machine [Rankl, 2007] [Rankl et Effing, 2003]. Au lieu de stocker un agent associé à chaque utilisateur dans le serveur distant, l'utilisateur porte son agent. De cette façon, la carte à puce est à la fois un moyen de stockage et un système d'identification et de localisation. Lorsque l'utilisateur arrive à proximité d'un point d'accès, l'utilisateur est identifié, puis l'agent mobile migre de la carte à puce sans contact au point d'accès. Ce dernier affiche le chemin nécessaire qui mène l'utilisateur à sa destination. Puis, l'agent mobile revient à la carte à puce.

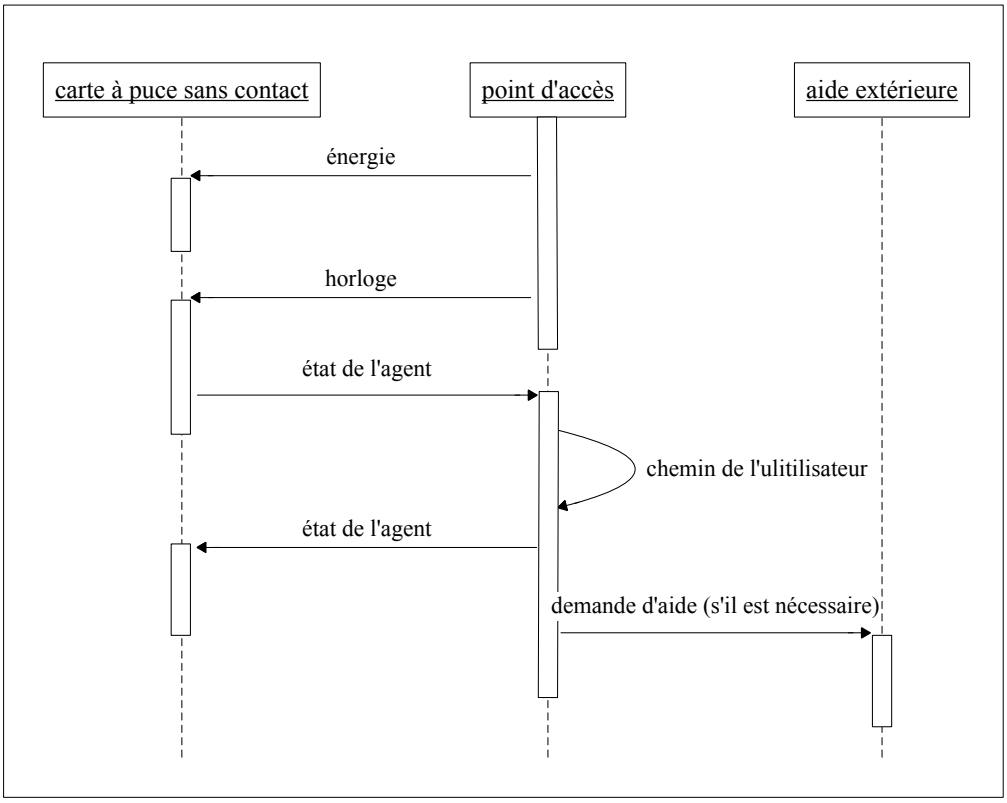


Figure 6.5 Migration d'état d'exécution d'agents par carte à puce RFID (sans contact)

Le paradigme d'agents mobiles permet l'autonomie et améliore la performance de notre système de guidage. Grâce à la technologie des cartes à puce sans contact, l'agent mobile interagit avec l'utilisateur d'un point d'accès à l'autre tout au long du trajet même si la connexion réseau tombe en panne. En effet, le système de guidage a seulement besoin du réseau pour mettre à jour les points d'accès, ou envoyer un message au serveur quand un utilisateur est incapable d'atteindre sa destination. Le choix de la troisième approche ne signifie pas qu'il est impossible de réaliser le système de navigation pédestre avec les deux

autres approches. En effet, la troisième approche offre plus d'avantages que les deux autres puisque la congestion que nous avons mentionnée peut être surmontée en utilisant un réseau ayant une bande passante plus élevée. Mais, cela augmentera le coût total du système.

Le choix de l'utilisation des cartes à puce sans contact est motivé non seulement pour des raisons de commodité mais aussi pour des raisons techniques. Les problèmes de contact entre les cartes à puce (avec contact physique) et le terminal (le point d'accès) qui constituent une des sources les plus fréquentes de l'échec dans les systèmes électromécaniques sont ainsi élégamment évités par les cartes à puce sans contact [Rankl et Effing, 2003].

Nous avons choisi la troisième option qui combine le paradigme d'agents mobiles et la technologie de cartes à puce sans contact. À notre avis, cette option est la plus adaptée à notre système de guidage. Les points faibles des cartes à puce sans contact sont sa mémoire de taille limitée et son taux de transfert entre la carte et le point d'accès qui ne dépasse pas 11500 bauds.

6.3.2 Interactions entre l'utilisateur, l'agent et le point d'accès

Notre système de guidage opère dans un espace intelligent. Ce dernier peut être un environnement ordinaire intérieur ou extérieur équipé des dispositifs de navigation qui permettent de percevoir et d'interagir avec un utilisateur pour le guider vers sa destination.

Pour développer un système à base d'agents mobiles, il est nécessaire d'avoir une plateforme appropriée. Cette dernière est un environnement permettant d'accueillir et d'exécuter un agent mobile. Nous utilisons la plateforme d'agents mobiles pour systèmes embarqués que nous avons décrite dans les chapitres précédents. Rappelons que cette plateforme est basée sur un noyau temps réel.

Dans cette plateforme, un agent est une tâche, donc un programme en exécution (un code et ses données). La mobilité de l'agent est définie comme suit:

(1) l'exécution de l'agent est interrompue sur le nœud courant (point d'accès) que nous appelons aussi le nœud source,
(2) les données représentant l'état de l'agent sont transférées du nœud source au nœud de destination,
(3) finalement arrivé au nœud de destination, l'exécution de l'agent continue là où elle a été interrompue sur le nœud source.

La plateforme d'agents mobiles offre le choix de transférer ou non le code de l'agent d'un point d'accès à l'autre selon les contraintes du système. Le code de l'agent étant disponible sur chaque point d'accès du réseau si la contrainte du système ne permet pas (un débit de transmission du réseau trop bas, par exemple). Malgré cette contrainte, à la différence des autres architectures, l'agent poursuit son exécution d'un point d'accès à l'autre grâce à la mobilité forte (les données et l'état).

Cependant, il reste à savoir si cette contrainte est raisonnable. Pour répondre à cette question, prenons le cas de notre système de navigation pédestre où chaque point d'accès utilise le même savoir-faire (code) qui implémente: les liens d'interaction, l'algorithme d'assignation de route, etc. Ce qui diffère d'un agent à un autre, c'est le contexte d'exécution et les données. Ainsi, il n'est pas nécessaire de charger le code de l'agent à chaque migration. Le contexte d'exécution et les données d'agents en circulation peuvent utiliser le même code disponible sur chaque point d'accès du réseau.

Au lieu de marcher avec un appareil de navigation tel que proposé par les auteurs dans [Chang et al., 2010] [Chang et al., 2009] [Chang et al., 2008], l'utilisateur porte une carte à puce sans contact durant son déplacement. Cette carte permet non seulement de contenir les données courantes de l'utilisateur mais également personnalise les interactions entre l'utilisateur et le point d'accès. Les données courantes sont ce qu'on appelle l'état d'exécution d'un agent, appelé aussi les données locales. En effet, les données sont le profil de l'utilisateur tels que l'identité, le chemin parcouru, la destination, la langue d'affichage, les capacités physiques et perceptives spécifiques, etc.

Lorsque l'utilisateur arrive à proximité d'un point d'accès, les données de la carte à puce sans contact migrent vers ce point d'accès. En utilisant un algorithme et la carte de l'endroit, le point d'accès indique à l'utilisateur le chemin nécessaire pour atteindre sa destination. Ensuite, les données courantes de l'utilisateur migrent du point d'accès vers la carte à puce sans contact. Le point d'accès contient les données et le code. Les données de l'utilisateur peuvent être divisées en données locales et globales. Le code est l'algorithme pour trouver le chemin nécessaire pour atteindre la destination. Les données globales constituent la carte de l'endroit. Les données locales sont les données courantes de l'utilisateur. Le code et les données globales sont fixes car tous les nœuds utilisent le même algorithme et la même carte de l'endroit. Ainsi, l'état de l'agent migre d'un point d'accès à l'autre selon les déplacements de l'utilisateur pour le guider. Notons que le code de l'agent n'est pas transféré car il est disponible sur chaque point d'accès.

Cependant, il y a un coût d'espace mémoire à la duplication du code. Mais, même les gros systèmes comme la plateforme Voyager utilisant un réseau à grand débit, le code de l'agent n'est pas toujours transféré vers le site de destination à chaque visite. En effet, le code est transféré seulement à la première visite du site. Si l'agent revient sur le site, la plateforme Voyager utilise le code de l'agent précédemment transféré. En revanche, on gagne un temps de transfert plus court puisque le coût de migration du code de l'agent éliminé. Il en résulte une exécution plus rapide de l'agent. La figure 6.6 montre la structure hiérarchique des composants du système de navigation.

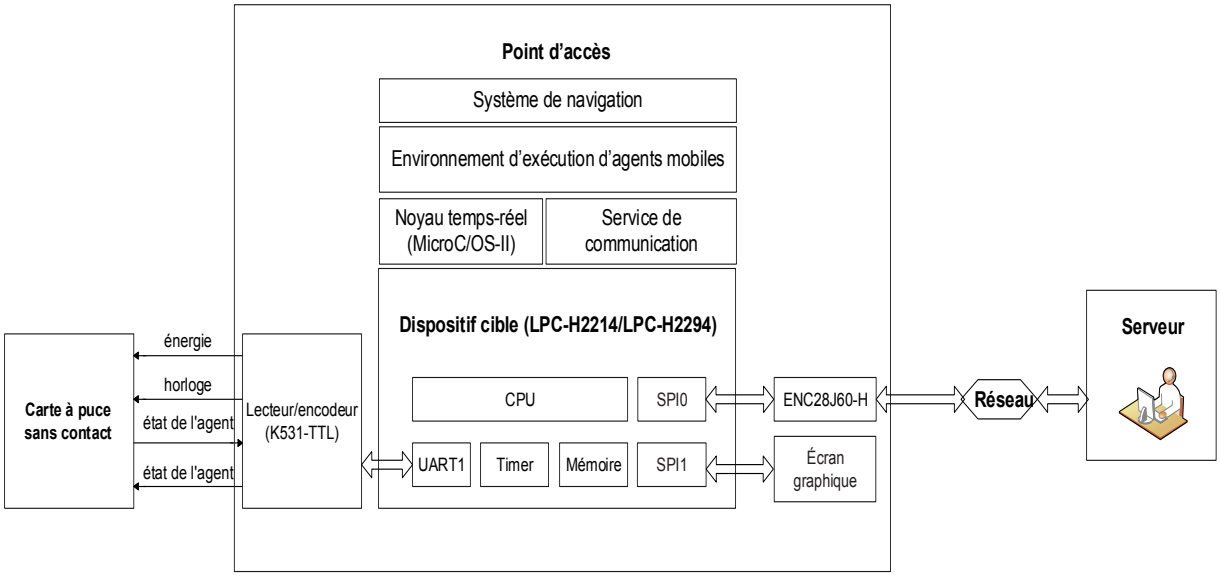


Figure 6.6 Structure hiérarchique des composants du système de navigation pédestre

Les points d'accès sont placés dans des endroits importants tels que les intersections, les sorties, les entrées des escaliers, les portes d'ascenseurs, etc. Ainsi, l'utilisateur consulte les points d'accès aussi souvent que nécessaire pour visualiser le chemin à prendre. Le système envoie un message à une aide extérieure lorsque l'utilisateur ne peut plus se rendre à sa destination. L'aide extérieure peut venir d'un membre de la famille de l'utilisateur ou d'un groupe de soutien. L'aidant exécute les actions nécessaires pour aider l'utilisateur.

6.3.3 Algorithme d'assignation de route

Pour assigner une route à un utilisateur, nous avons utilisé un algorithme permettant de trouver le chemin le plus court. Pour ce faire, nous avons construit un graphe de nœuds où chaque nœud représente un point d'accès et où chaque arc représente un chemin entre deux nœuds. Pour choisir une route entre une paire donnée des points d'accès, l'algorithme trouve le chemin le plus court entre eux sur le graphique. Dans notre système de navigation, la façon de mesurer la longueur du trajet est la distance en mètres entre le nœud source et le nœud de la destination.

Il existe plusieurs algorithmes pour trouver le chemin le plus court entre deux nœuds. Dans notre système, nous utilisons l'algorithme de Dijkstra [Dijkstra, 1959] pour trouver le chemin

le plus court. Cet algorithme calcule le chemin le plus court selon la combinaison de deux critères: la distance à parcourir et la capacité physique de l'utilisateur.

Cas d'utilisation

La figure 6.7 illustre un centre commercial pour décrire le cas d'un exemple d'utilisation de notre système de navigation. Chaque point d'accès du centre commercial affiche un message d'accueil en français et en anglais pour que les utilisateurs débutent leur déplacement avec la langue de leur choix. Ici, nous allons considérer le cas d'un utilisateur communiquant en français pour alléger le texte. Lorsque le point d'accès se trouve à une des entrées du centre commercial, il affiche «*Saisir le profil d'utilisateur*». Dans le cas contraire, le point d'accès affiche «*Approcher notre carte à puce au lecteur RFID*».

Supposons qu'un utilisateur entre son profil au point d'accès 7 (qui se trouve à une des entrées du centre commercial) et indique comme destination le nœud 9. Comme nous le montre la figure 6.7, il y a des escaliers entre les points d'accès 0 et 1, 7 et 8, 14 et 9, 16 et 11, 17 et 12, 4 et 5. En outre, il y a un ascenseur entre les points d'accès 15 et 10.

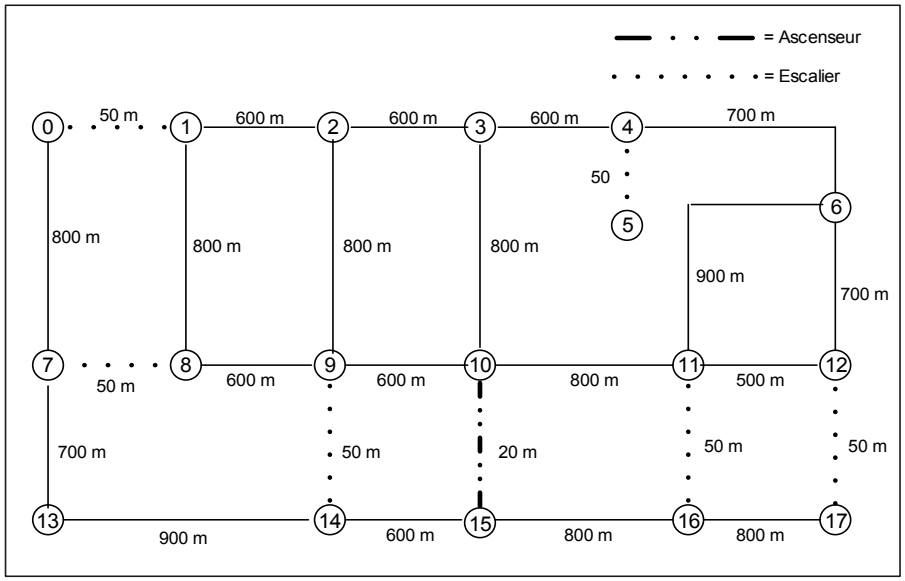


Figure 6.7 Exemple d'un graphe des nœuds (points d'accès)

Le système de navigation traite deux cas: un utilisateur non handicapé et une personne en fauteuil roulant. L'utilisateur non handicapé sera guidé à la route 7-8-9 (50 m + 600 m). En

revanche à cause de l'escalier, une personne en fauteuil roulant est incapable de passer directement de 7 à 9. Par conséquent, l'algorithme va proposer le parcours 7-13-14-15-10-9 (700 m + 900 m + 600 m + 20 m+ 600 m). Le point d'accès affiche une flèche vers le nœud 8 et un message correspondant s'il s'agit d'utilisateur non handicapé. Dans le cas d'un utilisateur en fauteuil roulant, le point d'accès affiche une flèche vers le nœud 13 et un message correspondant. Notons que les termes le point d'accès et le nœud sont interchangeables dans cette section puisqu'ils désignent la même chose. Sur chaque point d'accès, l'utilisateur visionne un symbole (une flèche, une image, etc.) et un message lui indiquant la direction à suivre.

Supposons que l'utilisateur n'a pas suivi exactement les indications et se trouve au point d'accès 11. Ce dernier va afficher une flèche vers le nœud 10 et un message correspondant. Lorsque l'utilisateur arrive au nœud 9, il affiche «*Vous êtes à votre destination*». Il est pertinent de préciser qu'une fois l'utilisateur est servi, le point d'accès affiche le message d'accueil pour les prochains utilisateurs. Notons que la figure 6.7 n'est pas à l'échelle.

6.4 Implémentation du système de navigation

Chaque point d'accès est composé d'un module microcontrôleur, un écran à cristaux liquides un lecteur/encodeur des cartes à puce sans contact. Une description détaillée des matériels utilisés se trouve dans l'annexe C de ce document. Le microcontrôleur, le lecteur/encodeur sans contact et le LCD graphique ont tous une interface TTL, ainsi leurs broches peuvent être connectées directement l'une de l'autre.

Chaque fois que l'utilisateur vient à proximité d'un point d'accès, le microcontrôleur lit les données de la carte à puce sans contact à l'aide du lecteur/encodeur. Ensuite, le microcontrôleur traite les données et affiche les instructions que l'utilisateur doit suivre pour atteindre sa destination sur le LCD graphique. Puis, le microcontrôleur transfère les données traitées vers la carte à puce sans contact à l'aide du lecteur/encodeur. Ces données sont l'état d'exécution de l'agent.

L'agent mobile peut adapter son aide selon le profil de l'utilisateur. Par exemple, si l'utilisateur a des problèmes de vision, le contraste de l'écran pourrait être ajusté dès que l'agent migre vers le point d'accès si le LCD le permet. L'agent mobile peut aussi adapter la langue d'affichage. Ceci est possible grâce à l'incorporation du profil de l'utilisateur dans l'état d'exécution de l'agent mobile.

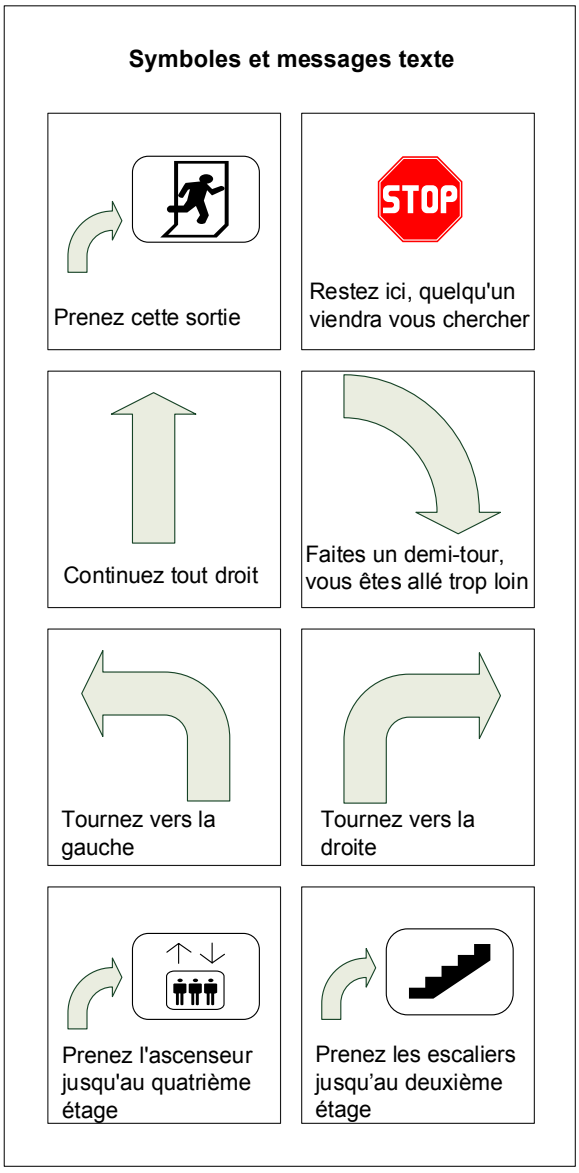


Figure 6.8 Combinaison des symboles et des messages texte

Comme le montre la figure 6.8, le système de navigation utilise une combinaison des symboles et des messages texte. Les symboles sont convertis en images bitmap et ensuite elles sont stockées dans la mémoire du microcontrôleur de chaque point d'accès. Ainsi, quand

un agent migre vers un point d'accès, le microcontrôleur affiche (à l'écran) l'image et le message appropriés pour guider l'utilisateur vers sa destination.

En outre, pour augmenter la sécurité, le point d'accès détecte également lorsque les utilisateurs ne parviennent pas à atteindre leur destination. Pour ce faire, le point d'accès utilise un certain nombre de paramètres tels que le nombre de nœuds visités, la durée cumulée des déplacements déjà effectués, la durée prévue pour atteindre sa destination.

6.4.1 Chargeur de code (programmateur de mémoire flash)

Comme mentionné précédemment, notre système n'a pas de chargeur de code. Alors, nous avons développé un programmateur de flash pour les microcontrôleurs LPC-H2214 et LPC-H2294. Le programmateur est chargé dans la mémoire flash interne du microcontrôleur et y réside en permanence. La figure 6.9 présente le chargement du code du système d'agents depuis le serveur vers le microcontrôleur LPC-H2214/LPC-H2294.

Le serveur est également utilisé pour mettre à jour les points d'accès. Une mise à jour est faite quand une nouvelle version du système est libérée ou lorsque des nouveaux points d'accès sont ajoutés.

Le chargement du code depuis le serveur se déroule comme suit. Le programmateur commence la communication en demandant le fichier à charger au serveur. Ce dernier envoie le fichier au programmateur. Le fichier est stocké dans la mémoire RAM externe avant l'écriture dans la mémoire flash externe pour des raisons d'efficacité: c'est plus rapide d'écrire dans la RAM que dans le flash. Comme la taille de notre système de navigation est inférieure à 1 mégaoctet, il faut une seule itération pour charger le code dans la mémoire flash externe.

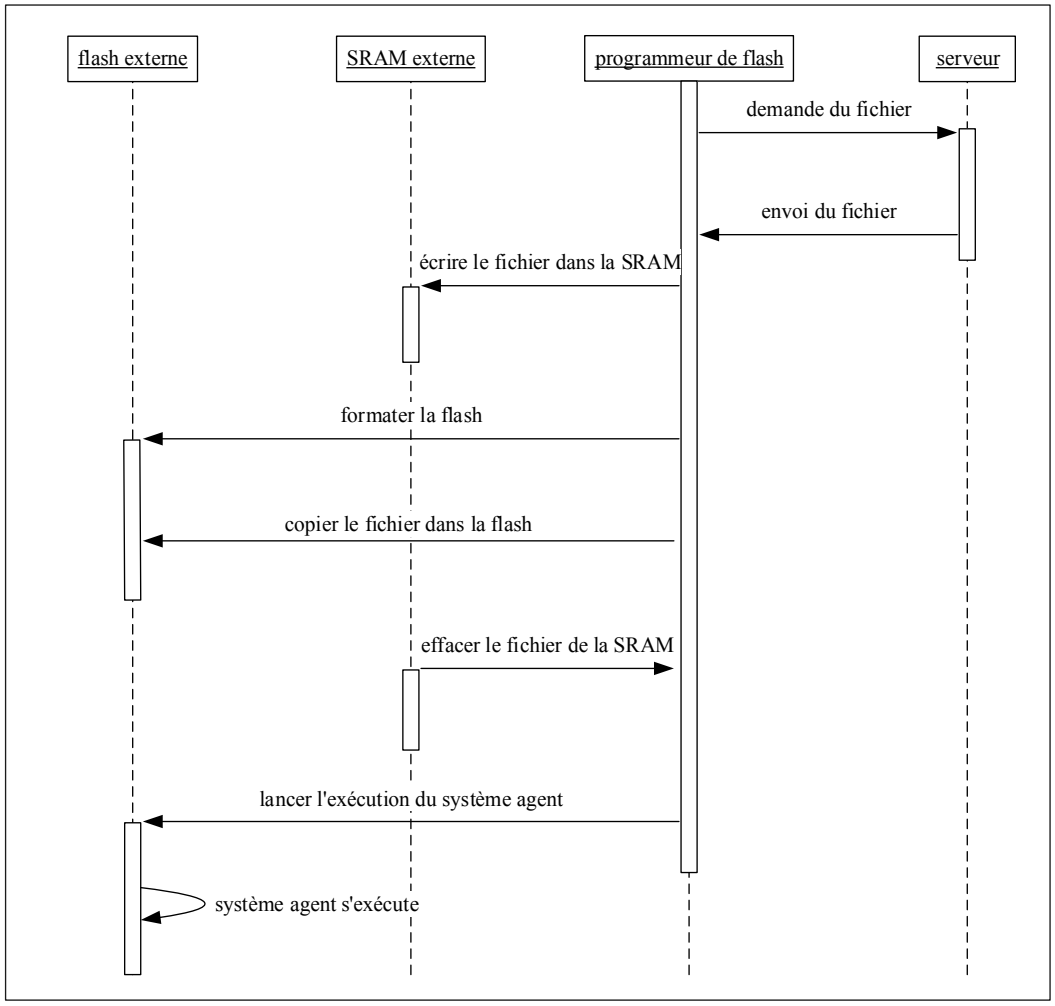


Figure 6.9 Chargement du code du système d'agents depuis le serveur vers LPC-H2214/LPC-H2294

La figure 6.10 présente les différents modules du système de navigation pédestre en mettant l'emphase sur leurs positions dans la mémoire du microcontrôleur LPC-H2214/LPC-H2294. Notons que le LPC-H2214 et le LPC-H2294 ont différents types de mémoire flash externe. Cela nécessite des traitements différents pour charger le fichier dans chaque type de flash. Le programmateur de flash réside dans le flash interne lorsque le système de navigation est en opération.

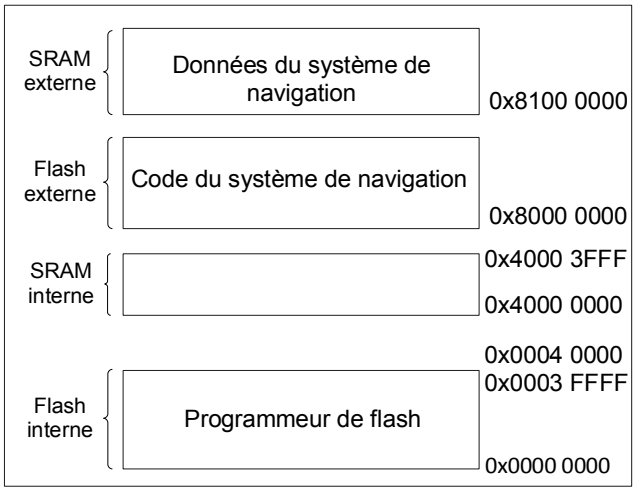


Figure 6.10 Système de navigation dans la mémoire du microcontrôleur LPC-H2214/LPC-H2294

6.4.2 Système de navigation pédestre

Dans notre système de navigation pédestre, l'utilisateur emploie un PC connecté au point d'accès pour construire son profil personnel. Avant d'entreprendre son itinéraire, chaque utilisateur établit son profil personnel qui comporte les éléments suivants: sa destination, son nom, son adresse, son numéro de téléphone, sa langue d'affichage, sa capacité physique, son mode de locomotion (par exemple, en fauteuil roulant ou non), ses éventuelles limitations sensorielles. Ainsi, la plateforme transforme cette fiche en agent mobile. Puis, l'utilisateur charge son agent dans sa carte à puce sans contact. Pour ce faire, il suffit tout simplement d'approcher sa carte à puce au point d'accès qui est équipé d'un lecteur/encodeur. Une fois que l'agent est chargé dans la carte à puce sans contact, le point d'accès le supprime afin d'accueillir l'agent d'un autre utilisateur. En effet, l'application qui met en œuvre notre système de navigation est multi-utilisateur.

L'utilisateur débute son itinéraire au nœud 1 pour, par exemple, se rendre au nœud 4. Pour chaque cas de configuration illustrée par la figure 6.11, nous avons expérimenté une dizaine d'utilisateurs ayant chacun sa propre carte à puce. Dans chaque cas d'utilisation, le système affiche le bon chemin que l'utilisateur doit prendre pour arriver à sa destination. Pour suivre le fil d'exécution de l'agent, un PC est connecté à chaque point d'accès. Le PC affiche tous

les descripteurs de la personne, l'historique (les nœuds visités) et la direction que l'utilisateur doit prendre. Pour guider l'utilisateur, l'écran affiche un des symboles et des messages texte.

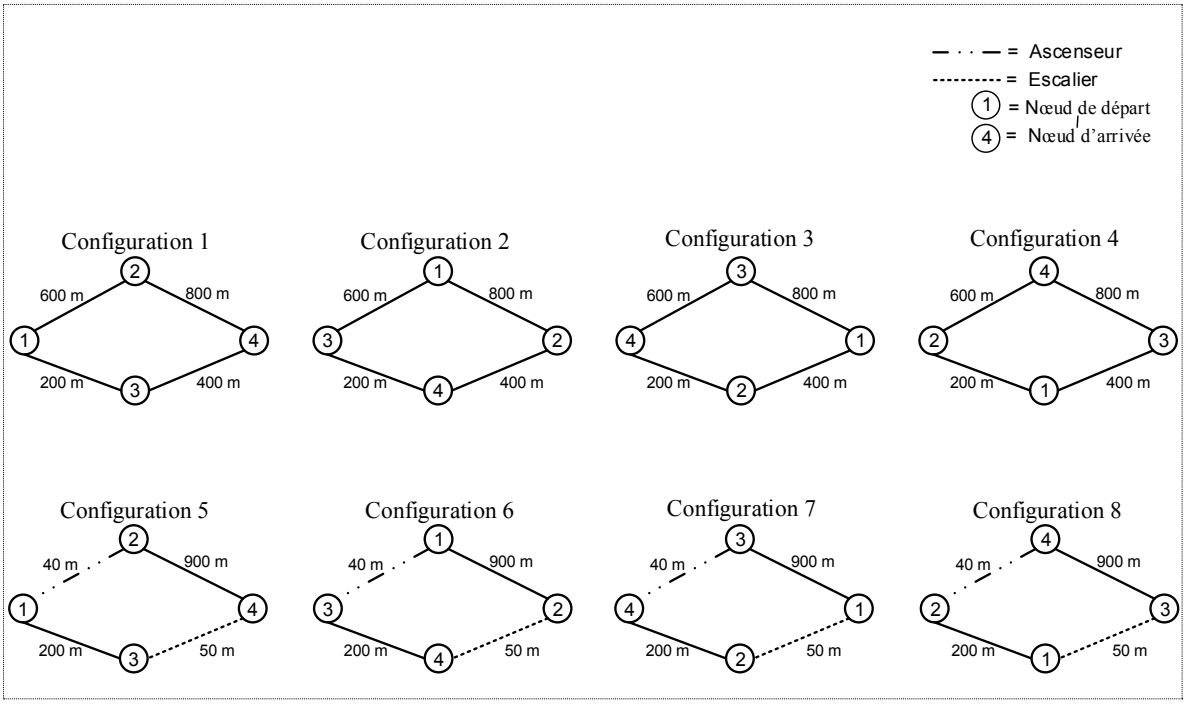


Figure 6.11 Différentes configurations du système de navigation pédestre

6.4.3 Comparaison de notre système de navigation avec des systèmes similaires

Bien que de nombreux travaux de recherche ont été effectués dans le domaine de la navigation pédestre, à notre connaissance, aucun n'a exploité la mobilité d'agents. Par conséquent, une comparaison quantitative directe entre notre système et les systèmes existants est difficile à obtenir puisque à notre connaissance nous sommes les premiers à concevoir un système pédestre à base d'agents mobiles. Cependant, nous pouvons présenter une comparaison qualitative entre notre système et les systèmes existants de navigation pour personnes ayant une déficience cognitive [Chang et al., 2010] [Chang et al., 2009] [Chang et al., 2008] [Liu et al., 2006]. Les systèmes existants utilisent exclusivement des dispositifs à main.

Contrairement au système dans [Liu et al., 2006], le dispositif à main proposé dans [Chang et al., 2010] permet de guider l'utilisateur sans une aide extérieure grâce aux tags RFID placés aux points de décision. Comme dans [Chang et al., 2010], le système de navigation pour personnes ayant une déficience cognitive proposé dans cette thèse utilise des tags RFID et des dispositifs de navigation permettant d'afficher le chemin que l'utilisateur doit prendre pour arriver à sa destination.

Notre système et celui dans [Chang et al., 2010] utilisent les mêmes technologies mais exploitent de manière opposée. En effet, la partie fixe de notre système, c'est-à-dire le dispositif de navigation, est mobile dans [Chang et al., 2010]. La partie mobile de notre système, c'est-à-dire les tags RFID et l'état d'exécution de dispositif de navigation, est fixe dans [Chang et al., 2010]. Dans notre système comme dans [Chang et al., 2010], le tag RFID et le dispositif de navigation doivent être proches l'un de l'autre à une distance de l'ordre de quelques centimètres pour visualiser le chemin à prendre pour arriver à la destination. En effet, les deux systèmes utilisent des tags RFID passifs. Dans notre système, les tags RFID passifs sont réalisés par des cartes à puce RFID.

Cependant, il reste à savoir quel système donne le plus d'avantages en termes d'utilisation et de coût total? Un premier avantage qu'a notre solution sur les systèmes existants de navigation pour personnes présentant une déficience cognitive demeure la portabilité des dispositifs de navigation. En effet, les utilisateurs n'ont pas besoin de porter de dispositifs de navigation puisqu'ils sont dissipés dans l'espace. Ainsi, les utilisateurs ne seront pas préoccupés par l'état de la batterie de leurs dispositifs de navigation puisqu'ils utiliseront l'électricité du site. Ceci pourra donc potentiellement réduire la charge cognitive des utilisateurs.

Pour avoir une idée du coût de notre système, imaginons un centre commercial équipé de 200 nœuds (dispositifs de navigation que nous appelons aussi des points d'accès) utilisés par 1000 personnes ayant une déficience cognitive. Le dispositif est composé d'un écran, d'un lecteur de tags RFID, des liens de communication, etc. Le prix d'un dispositif peut être estimé à $100. Comme, chaque utilisateur a besoin d'une carte à puce RFID, ceci donne 1000 cartes à

puce RFID. Le prix d'une carte à puce RFID peut être estimé à $2. Comme nous nous intéressons au rapport de coûts de deux systèmes, nous ne tenons pas compte des autres frais puisqu'ils s'annulent. Le coût total de notre système est estimé à:
= (nombre de points d'accès * prix à l'unité) + (nombre de cartes à puce RFID * prix à l'unité)
= (200 * $100) + (1000 * $2) = $22 000. Le tableau 6.1 présente le coût estimé de notre système de navigation pour personnes ayant une déficience cognitive dans différents scénarios.

Appliquons le même scénario précédent à ce système [Chang et al., 2010] aussi, c'est-à-dire 1000 personnes qui utilisent des tags RFID, pour estimer le coût. Rappelons que dans ce système, les dispositifs sont mobiles puisque chaque utilisateur se déplace avec le sien. Ceci donne 1000 dispositifs de navigation mobiles. En revanche, les tags RFID sont dissipés dans l'environnement, en occurrence dans le centre commercial. Ce système [Chang et al., 2010] nécessite 200 tags RFID fixes pour la localisation des utilisateurs. Le prix d'un tag RFID peut être estimé à $2. Le coût total de ce système [Chang et al., 2010] est estimé à:
= (nombre de dispositifs mobiles * prix à l'unité) + (nombre de tags RFID * prix à l'unité)
= (1000 * $100) + (200 * $2) = $100 400. Le tableau 6.2 présente le coût estimé du système dans [Chang et al., 2010] dans différents scénarios.

Tableau 6.1 Coût estimé de notre système dans différents scénarios

Scénario	Nombre de points d'accès	Prix de chaque point d'accès en $	Nombre de carte à puce RFID	Prix de chaque carte à puce RFID en $	Coût du système en $
1	40	100	100	2	4200
2	60	100	200	2	6400
3	80	100	300	2	8600
4	100	100	400	2	10800
5	120	100	500	2	13000
6	140	100	600	2	15200
7	160	100	700	2	17400
8	180	100	800	2	19600
9	200	100	1000	2	22000
10	220	100	1200	2	24400
11	240	100	1400	2	26800
12	260	100	1600	2	29200
13	280	100	1800	2	31600

Tableau 6.2 Coût estimé du système dans [Chang et al., 2010] dans différents scénarios

Scénario	Nombre de dispositifs mobiles	Prix de chaque dispositif mobile en $	Nombre de tags RFID	Prix de chaque tag RFID en $	Coût du système en $
1	100	100	40	2	10080
2	200	100	60	2	20120
3	300	100	80	2	30160
4	400	100	100	2	40200
5	500	100	120	2	50240
6	600	100	140	2	60280
7	700	100	160	2	70320
8	800	100	180	2	80360
9	1000	100	200	2	100400
10	1200	100	220	2	120440
11	1400	100	240	2	140480
12	1600	100	260	2	160520
13	1800	100	280	2	180560

La figure 6.12 présente le coût estimé de notre système (S1) versus celui dans [Chang et al., 2010] (S2). En effet, la figure 6.12 présente de façon graphique les résultats des tableaux 6.1 et 6.2.

Dans les mêmes scénarios, les résultats de calculs montrent que notre système de navigation pédestre réduit le coût de façon significative. En effet, la mise en œuvre de système de navigation pédestre à l'aide de notre solution de mobilité d'agents réduit le coût d'un facteur allant de 2.4 à 5.7 comme nous présente le tableau 6.3.

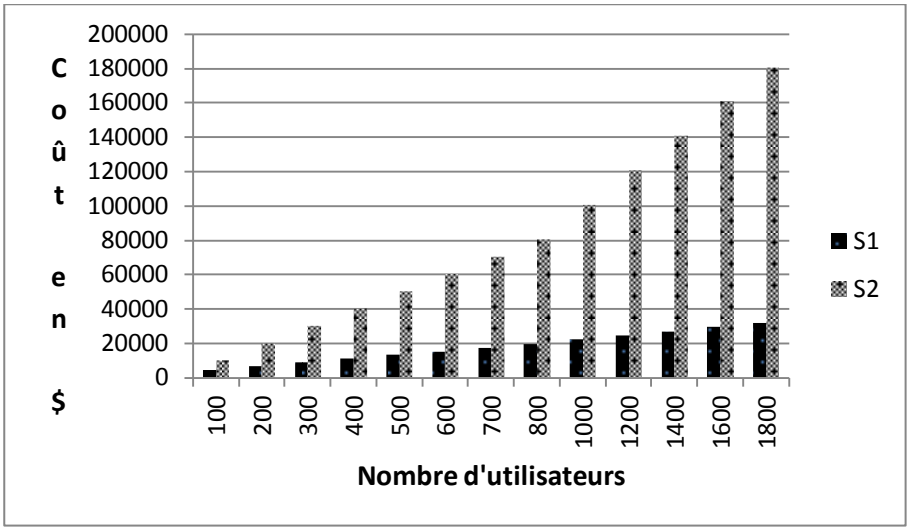


Figure 6.12 Coût estimé de notre système (S1) versus celui dans [Chang et al., 2010] (S2)

Tableau 6.3 Rapport de coûts entre les deux systèmes

Scénario	$\frac{\text{Coût estimé du système dans [Chang et al., 2010]}}{\text{Coût estimé de notre système}}$
1	2.400
2	3.144
3	3.507
4	3.722
5	3.865
6	3.966
7	4.041
8	4.100
9	4.564
10	4.937
11	5.242
12	5.497
13	5.714

Le coût du système proposé par [Chang et al., 2010] est essentiellement les dispositifs mobiles. Les utilisateurs peuvent-ils se servir de leur téléphone cellulaire? Oui, mais pas n'importe quel téléphone cellulaire. En fait, ils peuvent se servir des téléphones cellulaires dotés des lecteurs RFID. Ainsi, le coût de dispositifs mobiles peut être réduit.

Le coût de notre système est essentiellement les points d'accès. Ces derniers peuvent servir comme des panneaux publicitaires. Le coût de points d'accès peut être réduit également.

6.5 Conclusion

C'est dans le contexte de l'informatique omniprésente caractérisée par des objets communicants avec des fortes contraintes en ressources que notre solution de mobilité d'agents apporte pleinement sa contribution. Dans cette perspective, nous avons présenté différents exemples d'application où notre solution de la mobilité d'agents peut être exploitée, à savoir la domotique, la collecte d'informations en temps réel et la navigation pédestre pour personnes présentant une déficience cognitive.

Pour mieux évaluer la contribution de notre système, nous avons passé en revue les systèmes de navigation pédestre existants. Nous avons ensuite proposé des solutions alternatives pour personnes avec des problèmes de mémoire et d'orientation spatiale pour qui les systèmes existants de navigation présentent un défi. Dans cette perspective, nous avons développé un

système de navigation pédestre à base d'agents mobiles où l'utilisateur n'a pas besoin de porter un dispositif à main. Pour mettre en œuvre le système de navigation pédestre, nous avons exploité la solution de la mobilité forte d'agents proposée dans cette thèse.

Nous avons comparé notre système de navigation pédestre avec des systèmes similaires. Notre système apporte des avantages en termes d'utilisation et de coût. En effet, les utilisateurs n'ont pas besoin de porter de dispositifs de navigation puisqu'ils sont dissipés dans l'environnement. Ceci offre un autre avantage, les utilisateurs ne sont pas préoccupés par l'état de la batterie de leur dispositif de navigation. L'exploitation de notre solution de mobilité d'agents pour concevoir un système de navigation pédestre réduit le coût de façon significative par rapport aux systèmes existants qui proposent des dispositifs à main. En effet, dans différents scénarios d'un centre commercial, nos calculs montrent que l'exploitation de notre solution de mobilité d'agents réduit le coût par rapport aux systèmes existants [Chang et al., 2010] [Chang et al., 2009] [Chang et al., 2008] [Liu et al., 2006] d'un facteur allant de 2.4 à 5.7.

La bibliothèque qui met en œuvre notre système de navigation compte 1671 lignes de code. Celle-ci inclut un certain nombre de lignes de code pour le déverminage. Notons que le système de navigation est un état de prototype, donc un certain nombre d'optimisation peut être apportée. La mise en œuvre du système de navigation sous forme d'une bibliothèque permet la modularité, la réutilisation et ainsi la réduction de la taille de code.

Une limitation de notre système de navigation pédestre se situe au niveau de la capacité de mémoire des cartes à puce sans contact disponible (qui est de l'ordre de 4 à 16 kilooctets). Celle-ci implique que les informations qui constituent l'agent sont limitées et par conséquent, nous devons faire des choix pour garder les plus importantes. Toujours est-il qu'avec avec l'évolution technologique attendue, la capacité de mémoire augmentera considérablement.

CHAPITRE 7
CONCLUSION GÉNÉRALE

Dans ce dernier chapitre, nous dressons les contributions de ce travail et énumérons les perspectives qui pourront être poursuivies.

7.1 Contributions

Dans le cadre de cette thèse, nous avons proposé une architecture d'une plateforme d'agents mobiles natifs pour systèmes embarqués homogènes, baptisée µC/MAS. Nous avons réalisé un exemple d'utilisation de la plateforme non seulement pour vérifier et valider le bon fonctionnement mais également montrer l'utilité de notre solution de la mobilité forte d'agents dans le contexte de systèmes embarqués temps réel.

Dans un environnement homogène, non seulement le type de système d'exploitation doit être identique mais également le type de processeur et l'environnement d'exécution doivent être aussi. Pour ce faire, nous avons conçu:

- Une méthode de migration d'agents;
- Une directive de migration d'agents;
- Un format de transfert et une pile de protocole;
- Un mécanisme de migration d'agents par carte à puce RFID;
- Un système pédestre à base d'agents mobiles.

Les sections suivantes décrivent les contributions susmentionnées.

Méthode de migration d'agents

En exploitant l'analogie qui existe entre le changement du contexte d'exécution de tâches par un noyau temps réel et la mobilité d'agents, nous avons conçu une méthode de migration d'agents. L'originalité de cette méthode de migration est qu'elle peut être mise en œuvre dans n'importe quel système multitâche puisque nous avons exploité des mécanismes de bas niveau qui existent déjà dans ces systèmes.

Cette méthode de migration d'agents permet de lever la limitation qu'une tâche soit figée au système où elle commence son exécution. Grâce à la méthode de migration mise en œuvre dans le cadre de ce projet, une tâche peut être transformée en agent mobile et elle a ainsi la capacité unique de se déplacer d'un nœud à un autre dans le réseau.

Directive de migration des données

Dans son cycle de vie, un agent utilise différentes types des données. Pour gérer la migration, nous avons défini une directive visant à grouper les données en nous basant sur la classification des variables qui existe déjà en C/C++. La classe de stockage détermine l'emplacement en mémoire et la durée de vie d'une variable.

À l'aide d'un arbre binaire où chaque feuille spécifie la classe de stockage des variables ainsi que leur migration ou non avec l'agent, nous avons défini une directive de migration des données. Grâce à cette directive, les concepteurs des applications à base d'agents mobiles peuvent choisir la classe de stockage de données en vue de leur migration ou non avec l'agent.

Format de transfert et pile de protocole pour la migration d'agents

À l'aide de la norme XML en conjonction avec celle Intel HEX, nous avons défini un format de transfert et une pile de protocole pour la migration d'agents entre les nœuds. Pour ce faire, nous avons conçu deux analyseurs/encodeurs imbriqués afin de formater les données représentant l'agent.

Au nœud source, les données sont d'abord encodées en format Intel HEX, puis en XML. À la destination, les données suivent le traitement inverse grâce aux deux analyseurs/encodeurs imbriqués afin de restituer le fil d'exécution de l'agent dans une nouvelle tâche.

Mécanisme de migration d'agents par carte à puce RFID

Dans ce travail, nous avons introduit une nouvelle combinaison: le paradigme d'agents mobiles et la technologie des cartes à puce RFID. Grâce à cette combinaison, nous avons

conçu un mécanisme de migration de l'état d'exécution des agents mobiles par carte à puce RFID.

L'emploi des cartes à puce RFID comme moyen de transport de l'état d'exécution apporte une indépendance (autonomie) par rapport à tout système ultérieurement visité par l'agent et à toute utilisation du réseau. En effet, les données extraites de la carte à puce RFID contiennent toutes les informations nécessaires pour construire l'état d'exécution de l'agent. Lors de la récupération de l'état d'exécution, la carte à puce RFID et l'hôte communiquent par radio fréquence. Ils n'ont pas besoin donc de l'utilisation du réseau.

Système pédestre à base d'agents mobiles

Nous avons réalisé différents tests sur notre plateforme d'agents mobiles pour systèmes embarqués µC/MAS. Nous avons particulièrement réalisé un système de navigation pédestre non seulement pour vérifier et valider le bon fonctionnement mais également montrer l'utilité de notre solution de mobilité forte d'agents dans le contexte de systèmes embarqués.

Les systèmes pédestres existants utilisent exclusivement des dispositifs à main [Chang et al., 2010] [Chang et al., 2009] [Chang et al., 2008] [Liu et al., 2006]. Contrairement à ces systèmes, nous avons proposé un système de navigation pédestre à base d'agents mobiles. Ceci offre un certain nombre d'avantages. Un premier avantage qu'a notre solution sur les systèmes existants de navigation pour personnes ayant une déficience cognitive est la portabilité de dispositifs de navigation. En effet, les utilisateurs n'ont pas besoin de porter de dispositifs de navigation puisqu'ils sont dissipés dans l'environnement. Ceci offre un deuxième avantage: les utilisateurs ne sont pas préoccupés par l'état de la batterie de leurs dispositifs de navigation puisqu'ils sont dissipés dans l'environnement et utilisent l'électricité du site. Ceci peut potentiellement réduire la charge cognitive des utilisateurs. Un troisième avantage de notre système est la réduction du coût. En appliquant les mêmes scénarios, nos résultats de calculs montrent que notre système de navigation pédestre réduit le coût de façon significative par rapport aux systèmes existants de navigation pour personnes ayant une

déficience cognitive [Chang et al., 2010] [Chang et al., 2009] [Chang et al., 2008] [Liu et al., 2006].

Avantages et caractéristiques de la plateforme µC/MAS

Le principal avantage de la plateforme µC/MAS est qu'elle opère sur des machines avec des ressources très limitées. Notre plateforme peut opérer sur des machines ayant un espace de mémoire de l'ordre de quelques dizaines de kilooctets. Une caractéristique intéressante de cette plateforme µC/MAS est sa capacité de fonctionner sur des microcontrôleurs dépourvus d'une unité de gestion de mémoire. L'avantage essentiel de cette plateforme reste le fait qu'elle supporte la mobilité forte d'agents. En effet, à notre connaissance, il n'existe pas de systèmes d'agents mobiles permettant la mobilité forte sur des plateformes pour systèmes embarqués temps réel.

7.2 Perspectives

Dans le cadre de cette thèse, nous avons réalisé la plateforme µC/MAS qui est basée sur le noyau temps réel µC/OS-II. Chaque agent de cette plateforme est assigné à une priorité afin d'accéder à l'UCT. Pour tester la plateforme µC/MAS, nous avons rendu disponible la priorité de l'agent dans tous les nœuds du réseau. Qu'adviendra-t-il si la priorité de l'agent n'est pas disponible au nœud de destination? Une réponse à cette question se trouve dans l'adaptation de la plateforme µC/MAS aux noyaux temps réel supportant les tâches de même priorité. C'est le cas du noyau µC/OS-III qui est une évolution du µC/OS-II.

Dans ce nouveau noyau, les services offerts par conséquent les interfaces de programmation d'application (en anglais *Applications Programming Interface, API*) restent sensiblement les mêmes mais ont été enrichis avec des fonctionnalités supplémentaires. Il en résulte que les arguments, passés aux fonctions dans certains cas, ont radicalement changés par rapport à celles du noyau précédent. Les caractéristiques spécifiques du noyau temps réel µC/OS-III, en plus des celles de son prédécesseur, sont [Labrosse, 2011] [Labrosse, 2009]:

- Un ordonnancement en «Round Robin» entre les tâches de même priorité;
- Un nombre de tâches, de priorités et d'objets noyau illimité.

Au niveau système, il serait intéressant de définir une couche d'adaptation entre la plateforme µC/MAS et les autres systèmes qui ont les mêmes caractéristiques que celui utilisé dans cette thèse. En plus de celui susmentionné, nous pouvons citer les noyaux comme µClinux [Xiong et Gao, 2011] et eCos [Lohmann et al., 2006]. Ceci permet la circulation des agents mobiles entres les différents systèmes. Nous ne suggérons pas la création d'une machine virtuelle comme celle Java mais la définition d'une couche d'interface entre ces noyaux.

Il serait également intéressant de poursuivre cette extension en intégrant µC/MAS aux plateformes existantes comme Voyager. Cette intégration permet à la plateforme Voyager d'écrire un code natif et de le déployer sur des dispositifs de petite taille. Notons que la plateforme Voyager est un environnement multi-langage mais, dans son état actuel, seulement le composant écrit en Java supporte les agents mobiles et plus précisément la mobilité faible.

Au niveau application, on peut envisager de concevoir des robots qui changent de «corps» grâce à la technologie d'agents mobiles pour systèmes embarqués que nous avons développée dans le cadre de ce travail. On entend par «corps» toutes les composantes des robots qui ne sont pas logicielles: la carrosserie, les capteurs, le microcontrôleur, etc. La partie logicielle des robots est celle qui s'occupe du contrôle et de la prise de décision. Cette partie logicielle peut être mobile et, par conséquent, migrer d'un corps à un autre. Ceci n'est pas une transformation traditionnelle et elle pourra ouvrir une nouvelle voie de recherche grâce à notre travail.

ANNEXE A – PLATEFORMES D'AGENTS

A.1 Plateforme Voyager

Comme il est difficile de prévoir les exigences uniques de chaque client, Voyager est construit par des composants qui peuvent être étendus ou remplacés pour intégrer dans l'infrastructure existante. Voyager fait l'abstraction de la couche de système d'exploitation en fournissant un ensemble d'API universelle comme le montre la figure A.0.1 [Recursion Software, 2011].

Voyager intègre des composants pouvant s'exécuter aussi bien dans une machine virtuelle Java que dans un environnement «.net». Ainsi, Voyager devient une plateforme d'abstraction qui permet d'écrire un code natif dans le langage du choix de l'utilisateur (Java, C#, etc.) et de le déployer sur n'importe quel appareil quelle que soit la machine virtuelle du dispositif utilisé. Voyager supporte également plusieurs protocoles de communication tels que SOAP (Simple Object Access Protocol), IIOP (Internet Inter-ORB Protocol), etc. Voyager est un environnement multi-langage mais c'est seulement le composant écrit en Java qui supporte les agents mobiles et plus précisément la mobilité faible.

A.1.1 Modèle de programmation

Rappelons que la technologie des objets répartis vise entre autres à faire en sorte que la communication entre objets distants ressemble le plus possible, du point de vue du développeur, à une communication directe d'objet à objet.

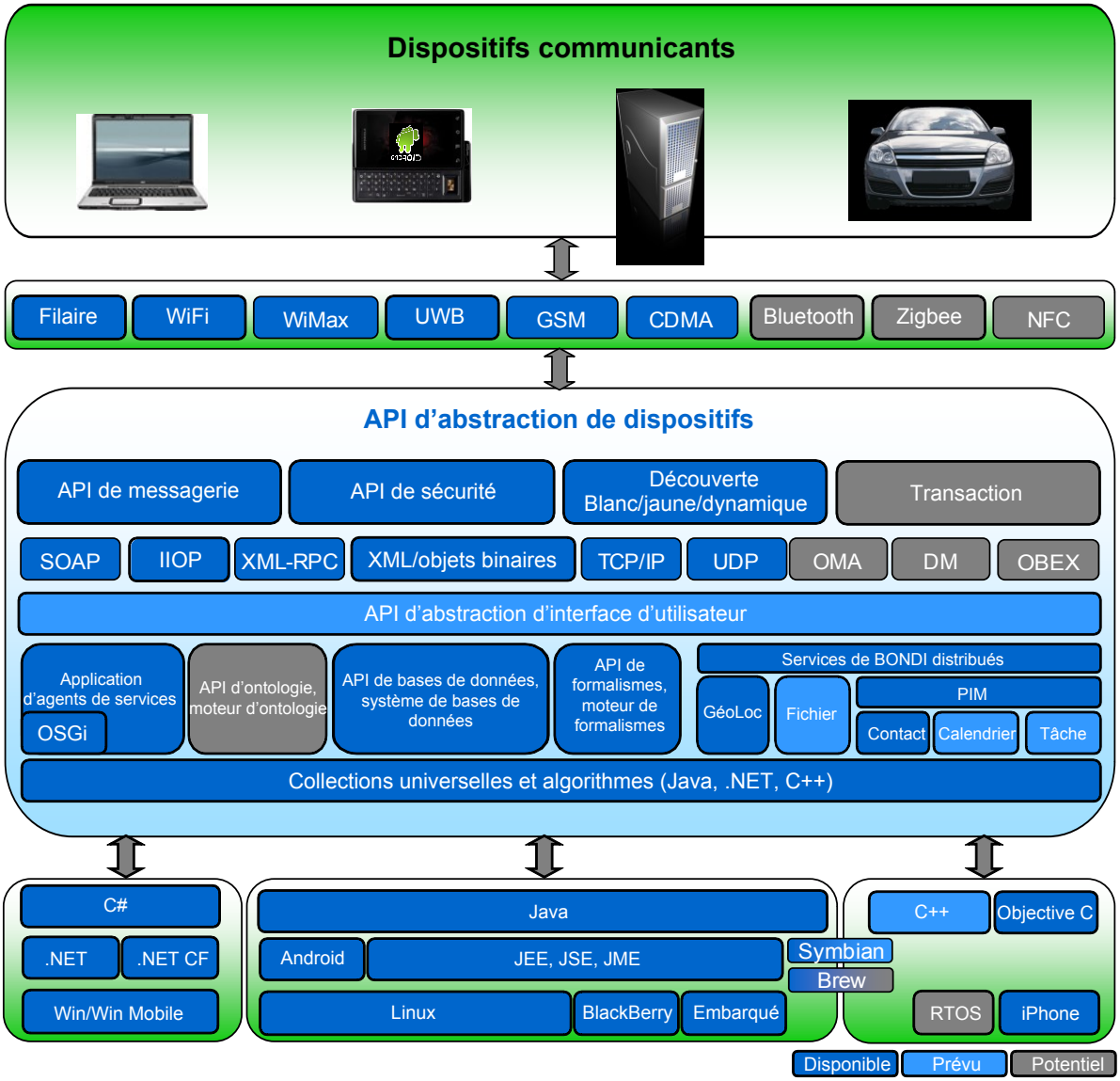


Figure A.0.1 Architecture de la plateforme Voyager [Recursion Software, 2011]

Voyager utilise pour cela le concept de proxy. Celui-ci est un objet qui agit comme référence vers un autre objet. Il sert entre autre à gérer ou contrôler les accès à l'objet final en agissant comme "ambassadeur" auprès d'un objet client. Cette technique est utilisée, dans le contexte des objets répartis, afin de fournir au client d'un objet distant un proxy local de cet objet, que le client peut ensuite utiliser pour communiquer avec l'objet à distance.

La clé consiste ici à faire en sorte que l'objet en question et son proxy implantent tous les deux la même interface. De cette façon, un programme client manipule de la même manière l'objet et son proxy et donc n'a pas besoin de différencier ces classes; l'interface commune suffit.

Ainsi, lorsqu'un client doit manipuler un objet distant, on ne lui fournit pas une référence directe à cet objet mais plutôt un proxy vers cet objet, généré dynamiquement par Voyager. Ce proxy est conçu de telle sorte que tout message lui étant transmis est routé grâce à l'ORB vers l'objet réel puis appelé via cet objet.

Voyager fait en sorte que les instances proxy qu'il génère dynamiquement implantent non seulement l'interface de leur objet primaire mais héritent aussi des fonctionnalités du classe générique proxy. Cette classe offre au programmeur quelques méthodes utilitaires telles qu'une méthode pour retourner l'URL de l'objet primaire et une méthode pour savoir si l'objet primaire est local ou non.

Avec Voyager, les interfaces utilisées comme spécialisation commune des objets et leur proxy sont toutes des interfaces au sens «Java» du terme. Elles ont par convention le même nom que leur objet primaire, auquel on ajoute un "I" comme première lettre. Ces interfaces peuvent être créées manuellement par le programmeur ou générées automatiquement grâce à un utilitaire fourni par Voyager.

Pour créer un objet réseau, c'est-à-dire un objet accessible via l'ORB Voyager, la méthode simple consiste à utiliser l'outil Factory.create (). Cette méthode permet de créer une nouvelle instance d'un objet sur toute machine ayant un ORB Voyager, d'exporter cet objet pour le rendre accessible aux appels à distance puis de retourner à l'appelant de la méthode un proxy vers cet objet.

Après que l'on ait obtenu un proxy vers un objet distant, on peut ensuite le manipuler comme s'il était local. On envoie un message sur le proxy, lequel transfère le message et ses paramètres (s'il y a lieu) vers l'objet via l'ORB. La valeur du retour, s'il y en a, est ensuite retournée à l'appelant et la méthode termine chez l'appelant.

Un ORB offre généralement un service d'identification (naming service) afin de donner des noms bien connus (et uniques) aux objets dans un système réparti. Un tel mécanisme est nécessaire dans la mesure où un programme client, lorsqu'il désire accéder à un objet réseau déjà existant, doit d'abord pouvoir identifier cet objet. On peut penser aussi à l'utilité d'un tel

service dans le cas où un programme client désire connaître quels sont les objets disponibles sur une autre machine et/ou dans un autre programme.

Dans Voyager, chaque ORB maintient une instance de "Directory", qui est une classe permettant d'associer des noms à des valeurs dans une structure hiérarchique de répertoires. La réunion de tous ces objets "Directory" (un par ORB) constitue l'espace réservé au service d'identification Voyager. Ainsi, un objet est identifié de façon unique par l'URL de base de la machine suivi du chemin complet de l'objet dans le "Directory" sur cette machine.

Dans Voyager, le service d'identification des objets est globalement accessible via les méthodes statiques de la classe *Namespace*. La méthode *bin()* permet d'assigner un nom à un objet tandis que la méthode *lookup()* retourne un proxy vers l'objet dont le nom est reçu en paramètre. On associe typiquement à un nom, non pas sa référence directe mais plutôt un proxy vers cet objet, puisque le but du service d'identification est justement de rendre accessible l'objet dans le système réparti. On note que l'utilitaire Factory.create () discuté plus haut permet de nommer un objet lors de sa création sans avoir recours directement à la classe *Namespace*.

Agrégation dynamique

L'agrégation dynamique est une étape fondamentale pour la modélisation d'objet et complète les mécanismes traditionnels de l'héritage et le polymorphisme. Ceci permet d'ajouter des données et du code à un objet de façon dynamique (au *runtime*). L'idée consiste à associer à l'objet primaire un ou plusieurs objets "secondaires" appelés *facettes*. L'objet primaire et l'ensemble de ses facettes, forment un agrégat et peuvent être ensuite considérés comme une seule entité, au niveau de la réclamation automatique de mémoire ou de persistance par exemple.

Voyager fournit un mécanisme qui permet d'associer de nouvelles facettes à un objet ou d'accéder à ses facettes existantes. Ce mécanisme est principalement accessible via la méthode statique of (*Objet*, *String*) de la classe *Facets*. Cette méthode vérifie si l'objet spécifié possède déjà comme facette une instance du type fourni en paramètre. Si ce n'est pas

le cas, une nouvelle facette de ce type est créée pour l'objet. Dans tous les cas, la méthode retourne à l'appelant un *proxy* vers la facette demandée, afin de lui permettre d'utiliser la fonctionnalité spécifique offerte par cette facette.

La mobilité d'un objet, dans *Voyager*, est en fait un comportement que l'on ajoute à cet objet par agrégation dynamique. La facette de mobilité d'un objet est toujours une instance de type *IMobility*. On peut obtenir en tout temps une instance de type *IMobility* pour un objet en appelant la méthode *of* () dans la classe utilitaire *Mobility* (qui appelle à son tour *Facets.of* ()). Grâce à la facette *IMobilily* ainsi obtenue, on est en mesure de déplacer l'objet primaire avec les méthodes *moveTo* () offertes par cette facette.

Agents mobiles

La différence fondamentale entre un agent mobile et un objet mobile, sur le plan conceptuel, est qu'un agent poursuit un objectif spécifique et qu'il possède les moyens lui permettant de l'atteindre sans aide extérieure. Ceci implique que l'agent mobile, contrairement à l'objet mobile, est responsable de ses propres déplacements et de sa propre autonomie.

Dans *Voyager*, un agent mobile est modélisé comme étant une facette d'un objet, cette facette étant de type *IAgent*. On associe généralement une instance de *IAgent* à l'objet principal constituant l'agent. Cette facette représente le côté "mobile" de l'agent global. L'interface d'un agent (*IAgent*) diffère de celle d'un objet mobile (*IMobility*) en ce sens qu'elle permet non seulement à l'agent de se déplacer vers une autre machine ou vers un autre objet via *moveTo* (), mais aussi de spécifier la méthode avec laquelle l'agent doit reprendre son exécution une fois que le déplacement est complété. En effet, les paramètres supplémentaires de la méthode *moveTo* (), de type *String* et *Object* [], représentent respectivement le nom de la méthode à exécuter sur la machine cible (ou destination) et la liste des paramètres devant être fournis à cette méthode. L'interface *IAgent* offre également un utilitaire permettant de contrôler "l'autonomie" d'un agent: la méthode *setAutonomous* (). Le terme "autonomie" a un sens particulier dans le contexte de *Voyager*. Le fait qu'un agent soit autonome signifie simplement qu'il ne sera pas détruit par le système de réclamation automatique de la mémoire même s'il n'existe plus aucune référence à cet agent ni à aucun de ses *proxys*. On note enfin

que *IAgent* fournit une méthode permettant de déterminer l'adresse du programme d'origine de l'agent.

Une facette de type *IAgent* peut donc servir comme base pour la création d'un agent mobile complexe, capable de se déplacer de lui-même sur le réseau et d'accomplir une tâche particulière sans dépendre d'interventions extérieures.

A.2 Plateforme Mobile-C

Mobile-C [Chen et al., 2006] [Chou et al., 2010] est une plateforme d'agents mobiles qui est mise en œuvre sous la forme d'une bibliothèque. Ainsi, elle peut être facilement intégrée dans des applications qui utilisent des agents mobiles. Cette plateforme est conforme aux normes FIPA et elle est destinée aux systèmes mécatroniques (mécanique/électronique) et aux systèmes embarqués «réseautés». Bien qu'elle soit une plateforme multiagents polyvalente, *Mobile-C* est conçue spécialement pour les systèmes temps réel et les machines avec des ressources limitées. Les agents mobiles dans un système de multiagent communiquent et travaillent en collaboration avec d'autres agents pour atteindre un but commun.

Pour des raisons de portabilité, les agents mobiles de *Mobile-C* sont écrits en C/C++. *Mobile-C* est intégré à un interprète appelé *Ch* [SoftIntegration, 2011] qui est un environnement d'exécution des agents mobiles en C/C++ [Chou et al., 2010]. *Ch* est un surensemble de C permettant l'écriture de script multiplateforme. Cet interprète supporte toutes les caractéristiques de la norme ANSI/ISO de 1990 de C et la plupart des nouveaux éléments ajoutés dans la norme ISO C99**,** tels que les nombres complexes, le tableau de longueur variable, les constantes binaires et le codage de virgule flottante de format IEEE 754. Il supporte aussi les classes, les objets et l'encapsulation en C++ pour la programmation à base d'objet.

Mobile-C est entièrement conforme aux normes FIPA tant au niveau de l'agent qu'au niveau de la plateforme. Au niveau de l'agent, la conformité comprend:

- un langage de communication entre agents,

- les protocoles d'échange de message,
- les actes communicatifs: chaque message est considéré comme une action communicative;
- les langages de représentation des connaissances pour décrire le contenu des messages.

Au niveau de la plateforme, *Mobile-C* fournit un système de gestion du cycle de vie d'agents, un canal permettant la communication entre agents sur le réseau et un répertoire de pages jaunes.

A.2.1 Architecture du système Mobile-C

La figure A.0.2 (extrait de la documentation de la plateforme [Mobile-C, 2011]) présente l'architecture du système *Mobile-C*. Les agences sont les composantes principales du système et sont installées dans chaque nœud de *Mobile-C*. Ils sont l'environnement d'exécution des agents stationnaires et des agents mobiles. Le cœur de l'agence est la plateforme d'agents qui fournit des services locaux pour les agents et des mandataires (*proxies*) pour accéder aux agences distantes. Une plateforme agent représente la fonctionnalité minimale requise par une agence pour supporter l'exécution d'agents. Les principales fonctionnalités de la plateforme agent peuvent se résumer comme suit:

- **Système de gestion d'agents** *(Agent Management System* ou *AMS)***:** AMS gère le cycle de vie des agents. Il contrôle la création, l'enregistrement, le retrait, la migration et la persistance d'agents. AMS maintient un répertoire d'identificateurs d'agents (*Directory of Agent Identifiers* ou *AID*) qui contient les adresses de transport (entre autres) des agents enregistrés. Chaque agent doit s'inscrire auprès d'un AMS afin d'obtenir un AID valide.
- **Agent de canal de communication** *(Agent Communication Channel ou ACC)*: ACC achemine les messages entre des entités locales et distantes, comprenant des messages utilisant un langage de communication d'agent (*Agent Communication Language* ou *ACL*). Ce service est responsable de toutes les interactions distantes entre les hôtes distribués. Le service de communication permet les interactions entre les agents, les agences et les entités non-agents.

- **Agent de gestion de sécurité** *(Agent Security Manager, ASM)*: ASM est responsable du maintien de politiques de sécurité pour la plateforme et les infrastructures telles que la communication et la sécurité au niveau de transport.
- **Services de répertoire** *(Directory Facilitator* ou *DF)*: Le DF sert des services de pages jaunes. Les agents peuvent enregistrer les services qu'ils fournissent à la communauté auprès de DF. Ils peuvent aussi consulter le DF pour trouver un service.
- **Environnement d'exécution d'agent** *(Agent Execution Engine ou AEE)*: AEE sert l'environnement d'exécution d'agents mobiles. Les agents mobiles doivent résider à l'intérieur de l'environnement pour s'exécuter. AEE doit être indépendant de la plateforme pour supporter l'exécution d'agents mobiles dans un réseau hétérogène.

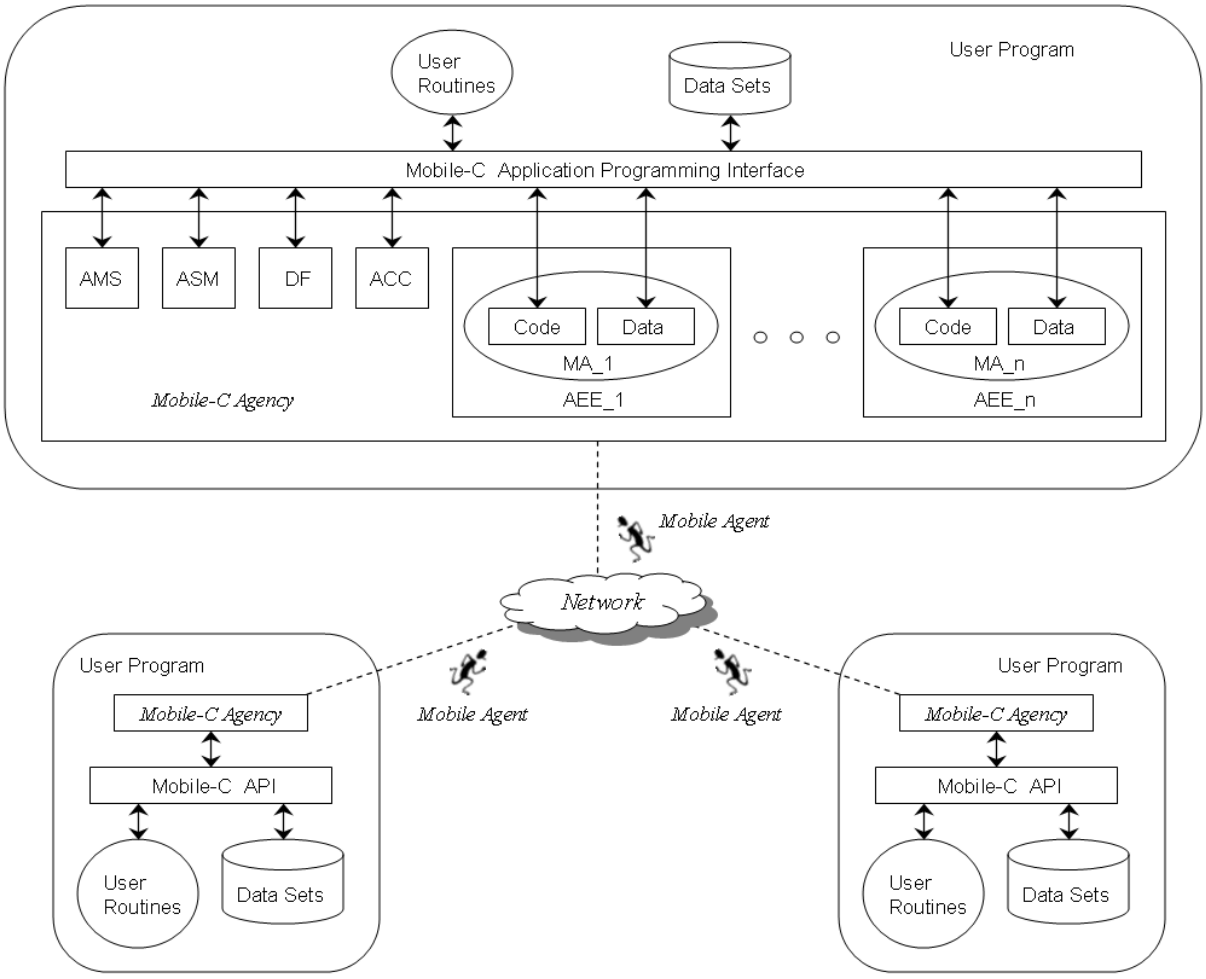


Figure A.0.2 Architecture de la plateforme Mobile-C [Mobile-C, 2011]

Support de la mobilité

Mobile-C met en œuvre la mobilité faible d'agents. L'exécution d'un code venant de l'extérieur sur un hôte introduit des problèmes de portabilité et de sécurités. Une approche commune pour ces problèmes est l'exécution des agents mobiles dans un environnement virtuel (un interprète ou une machine virtuelle).

Éléments essentiels de la plateforme *Mobile-C*

L'objectif premier de la bibliothèque (APIs) de *Mobile-C* est de fournir un mécanisme de mobilité et des communications à des programmes (agents) écrits avec un langage de bas niveau et plus particulièrement en C/C++. Les programmes écrits dans un langage de bas niveau ont l'avantage d'avoir un accès direct aux matériels, tels que la manipulation de la mémoire et des systèmes d'interface. Ces langages sont adaptés pour contrôler des matériels tels que des robots mobiles, des bras robotisés ou d'autres dispositifs électromécaniques. Les APIs facilitent plusieurs aspects essentiels des agents logiciels: la mobilité, les communications, la synchronisation, la latence et une faible empreinte mémoire (*small footprint*). La plateforme de *Mobile-C* est conçue pour:

- Migration de l'agent
    - Simplicité de la mobilité de l'agent: le processus de migration de l'agent est direct et facile à utiliser;
    - Faible surcoût de la migration d'un agent;
    - Rapidité du transfert de l'agent entre les nœuds;
- Normalisation du langage de communication d'agents FIPA-ACL
  La bibliothèque de *Mobile-C* supporte des messages conformes aux normes FIPA-ACL. Les agents de *mobile-C* peuvent utiliser ces messages pour communiquer les uns avec les autres, ainsi qu'avec d'autres agents des systèmes de FIPA tels que le JADE.
- Synchronisation entre agents et agent-agence
  Puisque la plateforme de *Mobile-C* a été conçue pour le contrôle du matériel et l'accès à la mémoire, de nombreuses méthodes de synchronisation ont été mises en œuvre pour coordonner les agents et les agences. Les paradigmes (ou variables) de synchronisation suivants ont été mis en œuvre:

- Exclusion mutuelle (*Mutual Exclusion*, *Mutex*);
- Variable condition (*Condition Variable*);
- Sémaphore;
- Barrière de synchronisation.

- Exécution du code d'agent mobile avec Ch, un interprète de C/C++

  Les agents *mobile-C* sont exécutés par un interprète C embarqué, connu sous le nom de *«Embedded-Ch»*. L'interprète *Ch* supporte un langage C augmenté appelé Ch. Ce dernier est un sur-ensemble de C99ANSI et contient quelques aspects de C++ ainsi que d'autres fonctions. En outre, l'interprète *Embedded-Ch* a sa propre API robuste et complète. L'interprète *Ch* est aussi multiplateforme, ce qui permet aux agents *Mobile-C* de s'exécuter sur de nombreux environnements hétérogènes. *Embedded-Ch* offre une fonctionnalité supplémentaire avec laquelle le programme C interprété peut lire la mémoire et appeler des fonctions de bas niveau pour accéder au matériel. Ceci est nécessaire pour des tâches telles que le contrôle de robots, ou les capteurs d'acquisition de données, etc.

- Faible empreinte mémoire

  Puisque la plateforme de *Mobile-C* a été conçue pour l'accès au matériel, elle dispose d'un très faible empreinte mémoire approprié pour les systèmes de petite taille (les systèmes embarqués), tels que le minuscule système de Gumstix [Huq et al., 2009] et les robots mobiles de K-Team Khepera [Ferretti et al., 2008]. Typiquement, un programme *Mobile-C* compilé statiquement avec toutes les fonctions activées utilise environ 500 Ko, il peut prendre un espace de mémoire de l'ordre de 6 Mo à exécution.

A.3 Plateforme Grasshopper

La plateforme *Grasshopper* [Breugst et al., 1998] est un environnement de développement et d'exécution pour agents mobiles. Cette plateforme permet une intégration du paradigme du client-serveur traditionnel et la technologie d'agents mobiles. La plateforme *Grasshopper* est la première plateforme conforme au standard MASIF [Guo et al., 2009]. Cette plateforme est mise en œuvre en Java et ne supporte que la mobilité faible.

A.3.1 Architecture de la plateforme

Grasshopper est constituée de plusieurs entités telles que des régions, des places, des agences et différents types d'agents.

Agence

Une agence est l'environnement d'exécution pour les agents mobiles et stationnaires. Chaque hôte doit posséder au moins une agence pour supporter l'exécution d'agents. Une agence *Grasshopper* est composée du cœur de l'agence et une ou plusieurs places. Le cœur de l'agence représente la fonction minimale requise par une agence pour supporter l'exécution d'agents. Il fournit les services de communication, d'enregistrement, de gestion, de transport, de sécurité et de persistance. La figure A.0.3 présente la structure hiérarchique des composantes de *Grasshopper*.

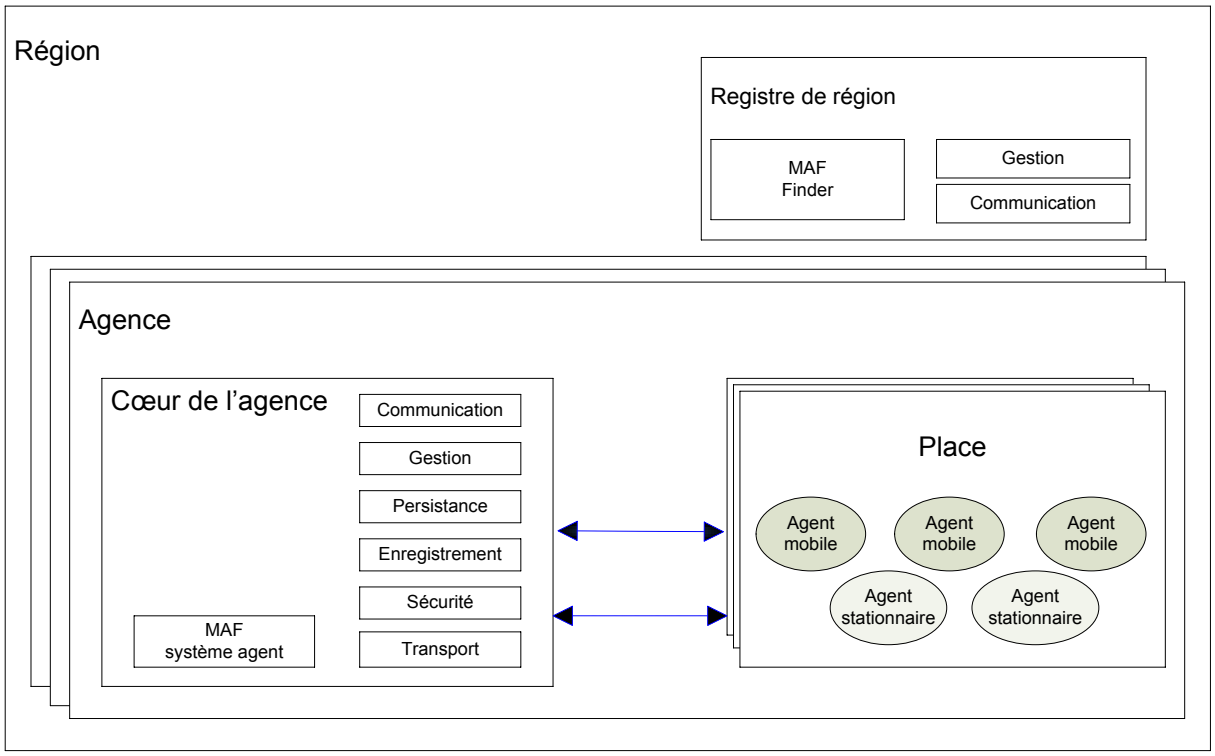


Figure A.0.3 Structure hiérarchique des composantes de Grasshopper [Grasshopper, 1998]

Place

Une place fournit un groupement logique de fonctions à l'intérieur d'une agence. Le nom de la place devrait refléter son but. Par exemple, dans chaque agence il existe par défaut une place nommée "InformationDesk". L'agent qui n'a pas de place est envoyé au "InformationDesk" où il peut chercher une information supplémentaire.

Région

Le concept de région facilite la gestion des composants répartis dans l'environnement de *Grasshopper*, c'est-à-dire des agences, des places et des agents. Les agences et leurs places peuvent être associées à une région spécifique. Les agences ainsi que leurs lieux peuvent être associées à une région spécifique en les inscrivant dans le registre. Si un agent se déplace vers un autre emplacement, l'information correspondante est automatiquement mise à jour. Une région peut contenir toutes les agences appartenant à une société spécifique ou à une organisation facilitant ainsi sa gestion.

Un registre de région maintient l'information sur tous les composants qui sont associés à une région spécifique. Quand un nouveau composant (c'est-à-dire une agence, la place, ou l'agent) est créé, il est automatiquement enregistré dans le registre de région correspondant. Bien que les agences et leurs places soient associées à une région pour leur cycle de vie, les agents mobiles se déplacent entre des agences des différentes régions. L'emplacement actuel d'agents mobiles, c'est-à-dire l'agence et la place dans laquelle ils résident, est mise à jour dans le registre de région correspondant après chaque migration. En entrant en contact avec le registre de région, d'autres entités (incluant des agents et des utilisateurs) sont à tout moment capables de mettre des agents, des places et des agences résidant dans une région.

En plus, un registre de région facilite l'établissement de connexion entre des agences ou des agents. Par exemple, l'agent A qui veut communiquer avec l'agent B est capable d'établir une connexion juste en sachant l'identificateur de l'agent B. Le service de communication *Grasshopper* détermine automatiquement l'emplacement actuel de l'agent B en entrant en contact avec le registre de région et établit la connexion. Le même principe s'applique à la

migration d'agent: un agent est capable de migrer juste en sachant le nom de l'agence de destination. Le nom de l'hôte, le numéro de port et le protocole de transport de la destination sont automatiquement détectés par l'agence source en entrant en contact avec le registre de région.

Service de Communication

Ce service est responsable de toutes les interactions distantes qui ont lieu entre les composants distribués de *Grasshopper*, comme la communication inter-agent, le transport d'agent et l'emplacement d'agents à l'aide du registre de région. Les interactions peuvent être effectuées via *CORBA IIOP*, Java *RMI*, ou les connexions de *plain socket*. Optionnellement, *RMI* et les connexions de *plain socket* peuvent être protégées à l'aide de s*ecure socket layer (SSL)* qui est le standard de sécurité Internet. Ce service supporte la communication synchrone et asynchrone, la communication multipoint et l'appel de la méthode dynamique. La figure A.0.4 illustre le support multi-protocole de la plateforme Grasshopper qui comprend les éléments suivants:

- *CORBA IIOP*: Ce protocole peut être utilisé dans tous les environnements qui supportent *CORBA* indépendamment d'une mise en œuvre *ORB* spécifique. Il utilise un mécanisme qui est conforme au standard *CORBA* pour se connecter à un objet utilisant un service de nommage. Notez que la communication *CORBA IIOP* est possible seulement si le client et le serveur supportent *CORBA*.
- *MAF IIOP*: Ce protocole est une spécialisation de *CORBA IIOP* développé pour l'interaction avec le système agent. Il est présenté dans la norme *MASIF* et fournit la connectivité entre les différents systèmes d'agents. Ainsi, *MAF IIOP* n'utilise pas le service de communication de *Grasshopper* et se connecte à l'interface de *MASIF* de l'agence directement. *MAF IIOP* a les mêmes prérequis que *CORBA IIOP*. Notez que la communication *MAF IIOP* est possible seulement si le client et le serveur supportent *CORBA*.
- *RMI*: *Java Remote Method Invocation (RMI)*, présenté dans le JDK 1.1, permet aux objets Java d'appeler les méthodes objets Java fonctionnant sur une autre machine virtuelle. Puisque ce protocole est inclus dans chaque machine virtuelle conforme JDK1.1, toutes les agences *Grasshopper* le supportent par défaut sans avoir besoin

d'une nouvelle installation ou d'un effort de configuration.

- *RMI* avec *SSL*: en utilisant ce protocole, *RMI* s'exécute au-dessus du protocole *secure socket layer (SSL)*. *SSL* fournit le transport sécurisé des données. Pour exécuter ce protocole, l'utilisateur doit probablement avoir un progiciel de sécurité complémentaire.
- *Plain Sockets* (en français, interface de connexion standard): c'est le protocole d'interactions distantes le plus rapide, en spécifiant un port de communication de l'hôte cible. Ce protocole est robuste et évite la surcharge des systèmes repartis. La communication *plain socket* est possible dans chaque environnement et constitue le protocole par défaut utilisé par les agences *Grasshopper*.
- *Plain Sockets* avec *SSL*: En utilisant ce protocole, les connexions *plain socke*t sont protégées par *SSL*. Les conditions préalables pour l'utilisation sont les mêmes que celles mentionnées par *RMI/SSL*.

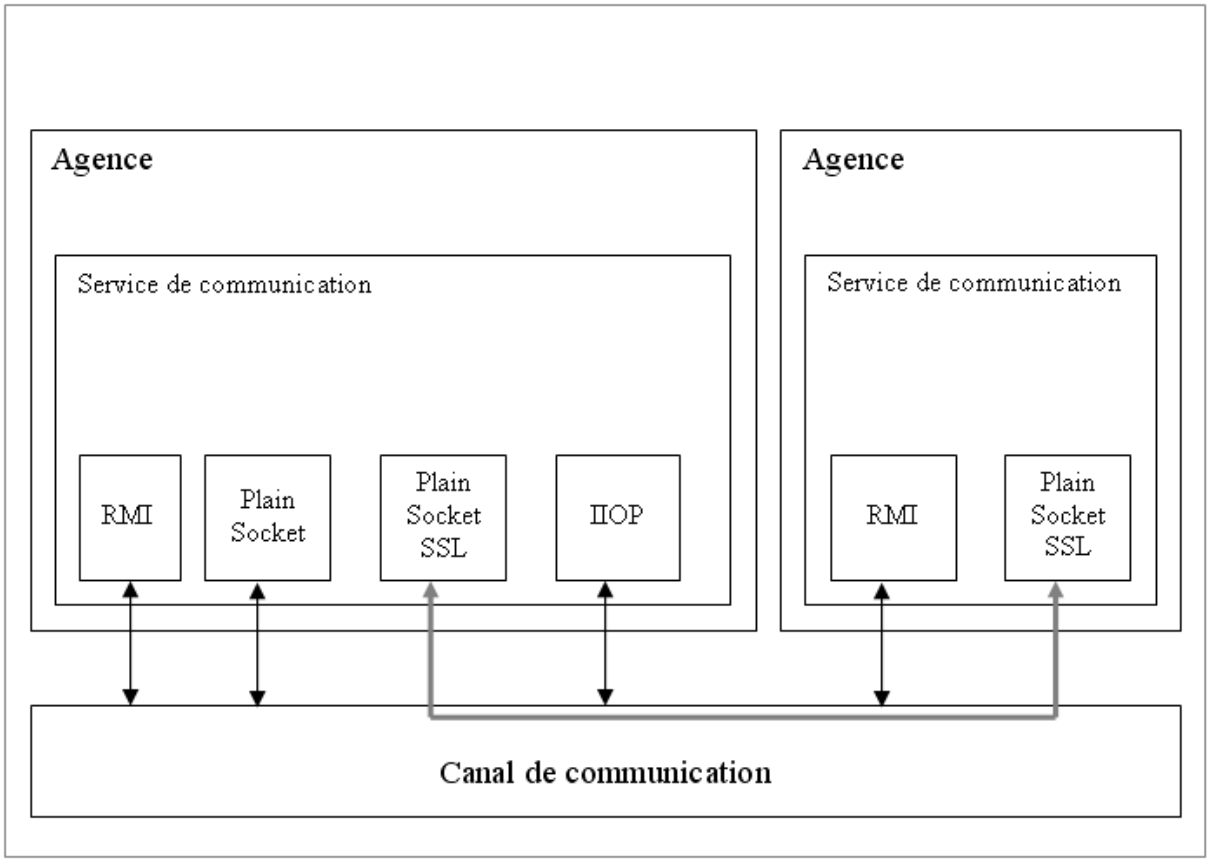


Figure A.0.4 Communication multiprotocole

Service d'enregistrement

L'agence doit savoir quels sont ses agents et ses places, d'une part pour des raisons de gestion externes et d'autre part pour livrer l'information sur des entités enregistrées. En plus, ce service est connecté au registre de région qui maintient l'information d'agents, des agences et des places dans une même région.

Services de gestion

Les services de gestion sont développés pour permettre la surveillance et le contrôle d'agents et des places d'une agence par des utilisateurs externes. Les fonctions suivantes sont supportées:

- Créer, supprimer, suspendre et reprendre des agents et des places;
- Obtenir l'information sur des agents et des places spécifiques;
- Inscrire tous les agents résidant en une place spécifique;
- Inscrire toutes les places d'une agence.

Service de transport

Ce service supporte la migration d'agents d'une agence à une autre. À l'agence de destination, l'agent reprend sa tâche depuis le début, tout en possédant les valeurs mises à jour de ses données, donc une migration faible. Le service de transport manipule l'externalisation et l'internalisation d'agents et la coordination du transfert réel qui est exécuté par le service de communication.

Service de sécurité

Grasshopper supporte deux types de sécurité:

- *La sécurité externe* qui protège les interactions à distance entre les composants *Grasshopper* répartis, c'est-à-dire la communication entre le registre de région et les agences. À cette fin, les certificats *X.509* et *secure socket layer (SSL)* sont utilisés. *SSL* est à la fois un protocole et un standard industriel qui utilise la cryptographie tant symétrique qu'asymétrique. En utilisant *SSL*, la confidentialité, l'intégrité des données

et l'authentification mutuelle des deux interlocuteurs en communication peuvent être réalisées.

- *La sécurité interne* qui protège les ressources d'agence de l'accès non autorisé par les agents. En plus, elle est utilisée pour protéger les agents les uns des autres. Ceci est réalisé par l'authentification et l'autorisation de l'utilisateur. La sécurité interne dans la plateforme *Grasshopper* est principalement basée sur les mécanismes de sécurité fournis par le *JDK*.

Service de persistance

Ce service permet le stockage d'agents et des places, c'est-à-dire l'information interne est maintenue à l'intérieur de ces composants au moyen d'un mécanisme persistant. De cette façon, il est possible de récupérer les agents ou les places au moment opportun, par exemple lors de redémarrage après une panne du système. Il existe deux types de persistance:

- La persistance implicite: Les places sont automatiquement persistantes si le service de persistance est activé. Cela signifie qu'une place existe même après la fermeture de l'agence et qu'elle sera de nouveau disponible après la reprise de l'agence. Le propriétaire d'agence peut aussi permettre la sauvegarde automatique de tous les agents quand l'agence ferme. Notez que la persistance implicite n'est pas visible pour les programmeurs d'agents. Elle est configurée et activée par les administrateurs d'agence.
- La persistance explicite: Trois mécanismes peuvent être distingués:
    - Un agent est constamment stocké périodiquement après un certain intervalle de temps sans suspendre son exécution de tâche. L'intervalle de temps est spécifié par l'agent lui-même, puis, l'agent n'a pas à se soucier de la maintenance de son information, puisque le service de persistance s'en charge automatiquement. Ce mécanisme est utile pour permettre le rétablissement d'agents quand une agence est redémarrée, par exemple après une panne système.
    - Les agents peuvent ordonner au service de persistance de les arrêter après un certain temps d'inactivité c'est-à-dire après qu'ils n'ont pas été utilisés par d'autres entités pour une certaine période de temps. Cependant, les agents restent enregistrés dans l'agence de région pour qu'ils puissent être redémarrés si d'autres entités essaient de les accéder. Ce mécanisme sert à sauvegarder les

ressources d'agence.

- Les agents peuvent être stockés explicitement à tout moment de leur propre initiative ou de l'initiative d'un utilisateur, par exemple l'administrateur d'agence. Cette démarche est utilisée par exemple si l'agence doit être temporairement arrêtée.

Notez que l'environnement *Grasshopper* peut établir une communication même sans un registre de région. Cependant, dans ces cas les agents et les agences doivent posséder toute l'information qui est exigée pour les interactions distantes, comme le nom d'hôte, les numéros de port, les protocoles de communication, etc.

Si l'environnement *Grasshopper* établit une communication avec un registre de région (ce que l'on recommande fortement), le registre doit être lancé avant que la première agence ne soit créée. Le registre n'est pas capable d'enregistrer les agences qui ont été créées avant lui.

ANNEXE B – CLASSES DE STOCKAGE

Le C/C++ dispose d'un éventail de classes de stockage qui permet de spécifier le type de variables que l'on désire utiliser dans une application:

- *auto:* les variables de classe *auto* sont définies au sein d'une fonction ou d'un bloc. Toute variable locale est donc par défaut *auto*. La portée d'une variable *auto* est la fonction dans laquelle elle est définie. La variable n'existe en mémoire que durant l'exécution de la fonction ou du bloc dans lequel elle est définie. Lorsque toutes les instructions du bloc sont exécutées, la variable disparaît de la mémoire et sa valeur est automatiquement perdue. Si le bloc est de nouveau exécuté, la variable est recréée. Une variable *auto* n'a pas de valeur initiale par défaut. Si elle apparaît dans le membre de droite d'une égalité, elle doit avoir été initialisée soit par affectation soit explicitement lors de sa définition.
- *static*: cette classe de stockage permet de créer des variables dont la portée est la fonction ou le bloc d'instructions en cours, mais qui, contrairement aux variables *auto*, ne sont pas détruites lors de la sortie de ce bloc. À chaque fois que l'on rentre dans cette fonction ou dans ce bloc d'instructions, les variables statiques existeront et auront pour valeurs celles qu'elles avaient avant que l'on quitte. Leur durée de vie est donc celle du programme, et elles conservent leurs valeurs. Si elle est initialisée lors de sa définition, elle ne sera plus réinitialisée lors d'un appel ultérieur. Un fichier peut être considéré comme un bloc. Ainsi, une variable statique d'un fichier ne peut pas être accédée à partir d'un autre fichier. Cela est utile en compilation séparée.
- *register:* les variables *register* obéissent aux mêmes règles que les variables automatiques, mais elles ne sont pas toujours rangées en mémoire de travail. Si le compilateur le peut, il les stocke dans des registres c'est-à-dire dans des zones de mémoire incluses dans le processeur. Si aucun registre n'est disponible, la variable recevra la classe auto. L'opérateur & ne peut pas être utilisé sur des variables registre. L'avantage d'avoir une variable conservée dans un registre réside avant tout dans la diminution du temps d'accès à cette variable en comparaison du temps d'accès à une

variable située dans la mémoire RAM. Ceci peut être intéressant lorsqu'une variable est souvent demandée.

- *volatile*: cette classe de variable sert lors de la programmation système. Elle indique qu'une variable peut être modifiée en arrière-plan par un autre programme (par exemple par une interruption, par un processus, par un autre processus, par le système d'exploitation ou par un autre processeur dans une machine parallèle). Cela nécessite donc de recharger cette variable à chaque fois qu'on y fait référence dans un registre du processeur, et ce même si elle se trouve déjà dans un de ces registres (ce qui peut arriver si on a demandé au compilateur d'optimiser le programme).
- *extern*: cette classe est utilisée pour signaler que la variable peut être définie dans un autre fichier. Elle est utilisée dans le cadre de la compilation séparée.
- Il existe également des modificateurs pouvant s'appliquer à une variable pour préciser sa constance:
- *const:* ce mot-clé est utilisé pour rendre le contenu d'une variable non modifiable. En quelque sorte, la variable devient ainsi une variable en lecture seule. Attention, une telle variable n'est pas forcément une constante: elle peut être modifiée soit par l'intermédiaire d'un autre identificateur, soit par une entité extérieure au programme (comme pour les variables volatile). Quand ce mot-clé est appliqué à une structure, aucun champ de la structure n'est accessible en écriture.
- *mutable*: disponible uniquement en C++, ce mot-clé ne sert que pour les membres des structures. Il permet de passer outre la constance éventuelle d'une structure pour ce membre. Ainsi, un champ de structure déclaré mutable peut être modifié même si la structure est déclarée *const*.

Pour déclarer une classe de stockage particulière, il suffit de faire précéder ou suivre le type de la variable par l'un des mots-clés suivants: *auto*, *static*, *register*, etc. On n'a le droit de n'utiliser que les classes de stockage non contradictoires. Par exemple, *register* et *extern* sont incompatibles, de même que *register* et *volatile*, et *const* et *mutable*. Par contre, *static* et *const*, de même que *const* et *volatile*, peuvent être utilisées simultanément. Les variables globales qui sont définies sans le mot-clé *const* sont traitées par le compilateur comme des variables de classe de stockage *extern* par défaut. Ces variables sont donc accessibles à partir

de tous les fichiers du programme. En revanche, cette règle n'est pas valide pour les variables définies avec le mot-clé *const*. Ces variables sont automatiquement déclarées *static* par le compilateur, ce qui signifie qu'elles ne sont accessibles que dans le fichier dans lequel elles ont été déclarées. Pour les rendre accessibles aux autres fichiers, il faut impérativement les déclarer avec le mot-clé *extern* avant de les définir.

ANNEXE C – MATÉRIELS UTILISÉS

Le point d'accès est composé d'un module ARM7, un écran à cristaux liquides (ou en anglais *liquid crystal display*, *LCD*), un module *ENC28J60-H*, un lecteur/encodeur des cartes à puce sans contact avec une antenne et une interface TTL (K531-TTL). Le module ARM7 est conçu sur la base d'un microcontrôleur *LPC-H2214/LPC-H2294* [NXP, 2012]. La différence entre le *LPC-H2214* et le *LPC-H2294* est la taille de leur mémoire flash externe qui est respectivement 1 mégaoctet et 4 mégaoctets. Dans cette section, nous allons décrire seulement le module *LPC-H2214* puisque que le reste de leurs caractéristiques sont identiques.

Figure C.0.1 Différentes vues du module LPC-H2214 [Olimex_a, 2011]

Comme nous le montre la Figure C.0.1, ce petit module dispose de 2 connecteurs mâles 2 x 17 points au pas de 2,54 millimètres. Un connecteur JTAG permet sa programmation via un câble dédié. Il peut être programmé aussi via son port USB. Les caractéristiques du module *LPC-H2214* sont [Olimex_a, 2011]:

1. Processeur: 16/32 bit LPC2214 (ARM7TDMI-S™);
2. 256 kilooctets de mémoire Flash;
3. 16 kilooctets de RAM et un bus mémoire externe;
4. 1 mégaoctet de SRAM (256 K x 32 bits) et 1 mégaoctet de Flash (512 K x 16 bits) présents au dos de la platine;
5. Convertisseur USB <-> Série intégré à la platine;
6. RTC, 4 x ADC (10 bits), 2 x UARTs, I2C, SPI, 2 x timers 32bit, 7 x CCR, 6 x PWM, WDT;

7. Fréquence d'opération jusqu'à 60 MHz;
8. Entrées/sorties compatibles avec les signaux de 5 V;
9. Connecteur JTAG (2 x 10 broches - compatible ARM-JTAG);
10. 2 étages de régulation intégrés de 1,8 et 3,3 V jusqu'à 800 mA;
11. LEDs de signalisation (présence d'alimentation et statut);
12. Capacité de filtrage sur l'alimentation;
13. Cavaliers pour "DBG (JTAG), BSL (bootloader) et JRST (RESET externe via RS-232)";
14. Quartz 14,7456 Mhz sur support (remplaçable);
15. Bouton-poussoir RESET;
16. Alimentation via le port USB (ou via alimentation externe de 5 Vcc);
17. Dimensions: 86 x 55 mm.

Figure C.0.2 Module K531-TTL OEM [SpringCard, 2009]

Le module *K531-TTL OEM* (*OEM* pour *Original Equipment Manufacturer*) est un lecteur/encodeur des cartes à puce sans contact de la norme *ISO/IEC 14443 A/B*. Il peut opérer avec tout microcontrôleur ou ordinateur hôte via son interface de communication série. La Figure C.0.2 illustre le module *K531-TTL OEM* [SpringCard, 2009]. La norme *ISO/IEC 14443 A/B* spécifie les propriétés et les modes de fonctionnement des cartes à puce sans contact [Rankl et Effing, 2003]. La distance de fonctionnement du module K531-TTL est typiquement entre 5 et 8 cm [SpringCard, 2009]. La norme *ISO/IEC 14443* de type A et de

type B peut opérer dans la gamme de fréquence de 13.56 MHz, mais les deux types sont incompatibles l'un de l'autre [Rankl, 2007]. La figure C.0.3 présente une vue détaillée du module K531-TTL.

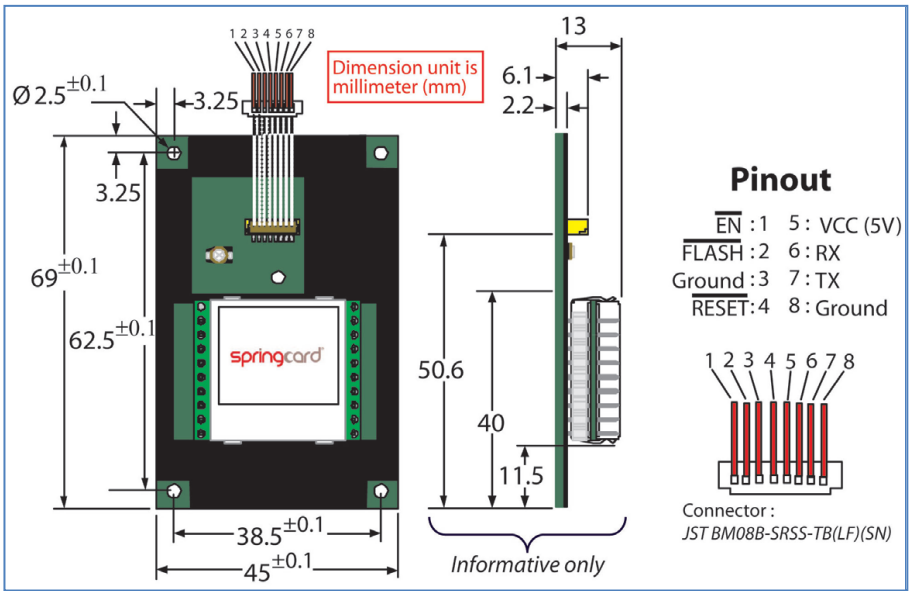


Figure C.0.3 Vue détaillée du module K531-TTL [SpringCard, 2009]

Pour afficher les symboles et les messages texte pour guider l'utilisateur, nous employons un LCD graphique *NHD-C12864EZ-fsw-ftw-P* [Newhaven Display, 2008]. Ce LCD possède 128X64 pixels et une interface de communication SPI (pour *Serial Peripheral Interface*).

Le module *ENC28J60-H* comprend essentiellement un contrôleur Ethernet *ENC28J60* associé à 2 LEDs (pour *Light-Emitting Diode*, en français diode électroluminescente) et un connecteur LAN (pour *Local Area Network*, connecteur RJ45). Il est conçu pour servir d'interface de réseau Ethernet à n'importe quel microcontrôleur équipé d'un SPI et répond aux spécifications du standard IEEE 802.3 [Microchip Technology, 2008].

LISTE DES PUBLICATIONS

M. A. Ibrahim et P. Mabilleau (mars 2012). Native Mobile Agents for Embedded Systems a été publié dans un chapitre d'un livre intitulé Embedded Systems - High Performance Systems, Applications and Projects, ISBN 978-953-51-0350-9, InTech, pages 195-220.

M. A. Ibrahim et P. Mabilleau (novembre 2009). Mobile agent-based system in a smart space for guiding people with spatial orientation problems a été publié dans Proceedings of the IASTED International Conference, Telehealth and Assistive Technology. Cambridge, Massachusetts, USA, pages 96-102.

M. A. Ibrahim et P. Mabilleau (février 2009). Mobile agent platform for embedded systems a été publié dans Proceedings of the IASTED International Conference. Innsbruck, Austria, pages 64-69.

LISTE DES RÉFÉRENCES

Acharya A., Ranganathan M., Sumatra , S. J. (juillet 1996). A Language for Resource-aware Mobile Programs. Dans 2nd International Workshop on Mobile Object Systems (MOS'96); Linz, Austria.

Attardi, G., Baldi, A., Boni, U., Carignani, F., Cozzi, G., Pelligrinini, A., Durocher, E., Filotti, I., Qing, W., Hunter, M., Marks, J., Richardson, C. et Watson, A. (novembre 1988). Techniques for dynamic software migration. Dans Proceedings of the 5th Annual Esprit Conference (ESPRIT'88), Volume 1, Bruxelles, Belgique, Pages 475- 491.

Attardi, G. et al., (1987). Specification for high Level Abstract Common Machine, Esprint Project N0 1228 (Chameleon). Dans Technical Report.

Babau, J.-P. (2005). Formalisation et structuration des architectures opérationnelles pour les systèmes embarqués temps réel. Dans la thèse de doctorat, l'Institut National des Sciences Appliquées de Lyon et l'université Claude Bernard de Lyon – LYON I, France.

Barak, A, Laden, O., et Yarom, Y. (1995). The NOW MOSIX and its Preemptive Process Migration Scheme. Dans TIEEE TCOS.

Bellifemine, F., Caire, G., et Greenwood, D. (2007). Developing Multi-Agent Systems with JADE. Chichester, England: John Wiley & Sons Ltd.

Bennani, M. T. (juin 2005). Tolérance aux fautes dans les systèmes répartis à base d'intergiciels réflexifs standards. Dans la thèse de doctorat, Institut National des Sciences Appliquées de Toulouse, France.

Bernett, M., Calvo , O., Wickert, S. (août 1987). Software for thee Fermilab smart Crate Controller. Dans IEEE Transactions on Nuclear Science, Vol. NS-34, No. 4.

Bershad, B. N., Zekauskas, M. J., Sawdom, W. A. (février 1993). The Midway Distributed Shared Memory System. IEEE International Computer Conference (COMPCON'93).

Bettini, L., De Nicola, R. et Loreti, M. (janvier 2002). Formalizing properties of mobile agent systems. Dans Coordination Models and Languages, pages 72–87.

Bidan, C. et Issarny, V. (octobre 1995). Un aperçu des problèmes de sécurité dans les systèmes informatiques. Publication interne nPPoPP 959, IRISA, campus universitaire de Beaulieu, 35042 Rennes cedex.

Birell, A.D. et Nelson, B. J. (février 1984). Implementing Remote Procedure Calls. Dans ACM Trans, On Computing Systems, Vol. 2, nPPoPP 1, p. 39-59.

Bohoris, C., Liotta, A., Pavlou, G. (2000). Evaluation of Constrained Mobility for Programmability in Network Management. Dans Services Management in Intelligent Networks, Proceedings of the 11th IEEE/IFIP International Workshop on Distributed Systems: Operations and Management.

Bouchenak, S., Hagimont, D. Krakowiak, De Palma, S. N., Boyer, F. (avril 2004). Experiences Implementing Efficient Java Thread Serialization, Mobility and Persistence. Software - Practice & Experience, Volume 34.

Bouchenak, S, Hagimont, D. (mai 2002). Zero Overhead Java Thread Migration. Dans INRIA Technical Report No. TR-0261, France.

Bouchenak, S. (octobre 2001). Mobilité et Persistance des Applications dans l'Environnement Java. Dans la thèse de doctorat d'Institut national polytechnique de Grenoble, France.

Bouchenak, S., Hagimont, D. (juin 2000). Pickling Threads State in the Java System. Dans Technology of Object-Oriented Languages and Systems Europe (TOOLS Europe'2000); Mont-Saint-Michel/Saint-Malo, France.

Breugst, M., Busse, I., Covaci, S. et Magedanz, T. (1998). Grasshopper -A Mobile Agent Platform for IN Based Service Environments. Dans Proceedings of IEEE IN Workshop, pages 279-290.

Cabillic, G. et Puaut, I. (janvier 1997). Stardust: An environment for parallel programming on networks of heterogeneous workstations. Dans Journal of Parallel and Distributed Computing, Vol. 40, nPPoPP 1, p. 65-80.

Campbell, R., Islam, N., Madany, P. et Raila, D. (septembre 1993). Designing and Implementing Choices: an Object-Oriented System in C++. Dans Communications of the ACM.

Cardelli, L. (1984). The Amber Machine. Dans Technical Memo, AT&T Bell Laboratoires.

Cardelli, L. (1983). The Functional Abstract Machine. Dans Technical Memo, AT&T Bell Laboratoires.

Cazes, A., Dlacroix, J. (2011). Architecture des machines et des systèmes informatiques. Dans Dunos, Paris, France.

Carzaniga, A., Picco, G. P., Vigna, G. (2007). Is Code Still Moving Around? Looking Back at a Decade of Code Mobility. Dans 29th International Conference on Software Engineering (ICSE'07 Companion), IEEE Computer Society.

Chang, Y-J., Chu, Y-Y., Chen, C-N et Wang, T-Y. (2008). Mobile Computing for Indoor Wayfinding Based on Bluetooth Sensors for Individuals with Cognitive Impairments. Dans International Symposium on Wireless Pervasive Computing, Santorini, Greece.

Chang, Y.-J., Peng, S.-M., Wang, T.-Y., Chen, S.-F., Chen, Y.-R., Chen, H.-C.(2010). Autonomous indoor wayfinding for individuals with cognitive impairments. Dans Journal of neuroengineering and rehabilitation.

Chang, Y-J., Chen, Y-R., Chang, C. Y. et Wang, T-Y.(2009). Video Prompting and Indoor Wayfinding Based on Bluetooth Beacons: a Case Study in Supported Employment for People with Severe Mental Illness. Dans International Conference on Communications and Mobile Computing, Kunming, Yunnan, China.

Chang, Y.-J., Chen, C.-N., Chou, L.-D. et Wang, T.-Y.(2008). A novel indoor wayfinding system based on passive RFID for individuals with cognitive impairments. Dans 2nd International Conference on Pervasive Computing Technologies for Healthcare, Tampere, Finland.

Chen, B., Cheng, H. H. et Palen, J. (décembre 2006). Mobile-C: A Mobile Agent Platform for Mobile C/C++ Code. Dans Software - Practice & Experience, Vol. 36, Issue 15, pp. 1711-1733.

Chevochot, P. et Puaut, I. (novembre 1997). Tolérance aux fautes dans les systèmes d'exploitation temps réel à sûreté critique. Publication interne no 1142, IRISA, campus universitaire de Beaulieu, 35042 Rennes cedex.

Chhetri, M. B., Price, Krishnaswamy, R., S. et Loke, S.W (2006). Ontology-based agent mobility modelling. Dans Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS'06), Kauai, IEEE Computer Society.

Chou, Y.-C., Ko, D., Cheng, H. H. (février 2010). An Embeddable Mobile Agent Platform Supporting Runtime Code Mobility, Interaction and Coordination of Mobile Agents and Host Systems. Dans Information and Software Technology, Vol. 52, Issue 2, pp. 185-196.

Conti, J. P.(janvier 2007). Internet of things. Dans IET Communications Engineer.

Cubat dit cros, C. (décembre 2005). Agents Mobiles Coopérants pour les Environnements Dynamiques Agents Mobiles Coopérants pour les Environnements Dynamiques. Dans la thèse de doctorat de l'Institut National Polytechnique de Toulouse, France.

Czajkowski, G. (octobre 2000). Application Isolation in the Java Virtual Machine. Dans 17th Annual ACM Conference on Object-Oriented Programming, Systems and Languages (OOPSLA'00); Minneapolis, MN, USA.

De paoli, D. (1994). The Multiple Strategy Process Migration Manager for RHODOS: The Logical Design. Dans Technical report, Deakin University, Victoria, Australie.

Digi International (2009). Product Manual v1.xEx - 802.15.4 Protocol, XBee®/XBee-PRO® RF Modules. En ligne http://ftp1.digi.com/support/documentation/90000982_B.pdf (page consultée le 10 août 2013)

Dijkstra, E. (1959). A Note on Two Problems in Connexion with Graphs. Dans Numerische Mathematik, Vol. 1, pages 269-271.

Douglis, F. et Ousterhout, J. (août). Transparent Process Migration: Design Alternatives and the Sprite Implementation, Software—Practice and Experience, Vol. 21(8), 757–785.

Douglis, F. (octobre 1989). Experience with process migration in Sprite. Dans Technical report, University of California Berkeley, California.

El falou, S. (novembre 2006). Programmation répartie, optimisation par agent mobile. Dans la thèse de doctorat de l'Université de Caen, France.

Elloumi, W. (décembre 2012). Contributions à la localisation de personnes par vision monoculaire embarquée. Dans la thèse de doctorat de l'Université d'Orléans, France.

Elloumi, W., Leconge, R., Royer, E., Treuillet, S. (2011). Localisation pédestre: Synthèse bibliographique et illustration d'une approche par vision monoculaire embarquée. Dans ORASIS - Congrès des jeunes chercheurs en vision par ordinateur.

Engel, J. (1999). Programming for the Java Virtual Machine. Dans Addison Wesley.

Ertan, S., Lee, C., Willets, A., Tan, H., and Pentland, A. (1998). A wearable haptic navigation guidance system. Dans Proc, Second International Symposium on Wearable Computers, pages 164-165.

Ferber, J. (1999). Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence. Dans Addison Wesley Professional, 528 p.

Ferretti, E., Kiessling, R., Silnik, A., Petrino, R. et Errecalde, M. (2008). Integrating vision-based motion planning with defeasible decision making for the Khepera robot. Dans 5th International Conference on Electrical Engineering, Computing Science and Automatic Control.

Filgueiras, T. P., Lung, L. C., de Oliveira Rech, L. (2012). Providing Real-Time Scheduling for Mobile Agents in the JADE Platform. Dans 15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing.

Finkel, R. et Artsy, Y. (hiver 1989). The process migration mechanism of Charlotte. Dans IEEE Computer Society Technical Committee on Operating Systems Newsletter.

FIPA Agent Management Support for Mobility Specification (2000). En ligne http://www.fipa.org/specs/fipa00087/PC00087A.pdf (page consultée le 10 août 2013)

FrameIP TcpIP. SNMP En ligne http://www.frameip.com/snmp/ (page consultée le 15 février 2014)

Fuggetta, A., Picco, G. P., et Vigna, G. (mai 1998). Understanding Code Mobility. Dans IEEE Transactions on software engineering, Vol. 24, NO 5, p. 342-361.

Funfrocken, S. (septembre 1998). Transparent Migration of Java-based Mobile Agents (Capturing and Reestablishing the State of Java Programs). Dans 2nd International Workshop Mobile Agents 98 (MA'98); Stuttgart, Germany.

Geib, J., Gransart, C. et Merle, P.(Octobre 1999). CORBA: des concepts à la pratique. Editions Dunod, Paris, France.

Gershenfeld, N. (1999). When things start to think. Dans Henry Holt.

Gilliéron, P-Y., Daniela Büchel, D., Spassov, I., Merminod, B. (2004). Dans Indoor Navigation Performance Analysis, ENC GNSS.

Goscinski, A. (1994). Research of Distributed and Open Systems and Processing: The RHODOS Project. Dans Technical report, Deakin University, Australie.

Grasshopper (2001). A Universal Agent Platform Based on OMG MASIF and FIPA Standards. En ligne http://cordis.europa.eu/infowin/acts/analysys/products/thematic/agents/ch4/ch4.htm (page consultée le 16 août 2013)

Gray, R. S., Cybenko, G., Kotz , D., Peterson, R. A. et Rus, D. (2002). D'agents: Applications and performance of a mobile-agent system. Dans Softw., Pract. Exper., 32(6): 543–573.

Gray, R. S., Cybenko, G., Kotz, D. et Rus, D. (2001). Mobile agents: Motivations and State of the Art. Dans AAAI/MIT-Press.

Grosso, W. (octobre 2001). Java RMI. Dans O'Reilly Media.

Guo F., Zeng, B. et Cui, L.(2009). A Distributed Network Management Framework Based on Mobile Agents. Dans Third International Conference on Multimedia and Ubiquitous Engineering.

Hagimont, D., Boyer, F. (2001). A Configurable RMI Mechanism for Sharing Distributed Java Objects. Dans IEEE Internet Computing, 5(1).

Helal, A., Moore, S. et Ramachandran, B. (octobre 2001). Drishti : An Integrated Navigation System for Visually Impaired and Disabled. Dans Proceedings of the 5th International Symposium on Wearable Computer.

Henriksen, A. et Hansen, J. G. (décembre 2002). Nomadic Operating Systems. Dans le mémoire de maîtrise (Master's thesis), University of Copenhagen.

Hesch, J. A. et Roumeliotis, S. I. (avril 2007). An Indoor Localization Aid for the Visually Impaired. Dans Proc. 2007 IEEE International Conference on Robotics and Automation (ICRA'07).

Hohl, F. (1998). A Model of Attacks of Malicious Hosts against Mobile Agents. Dans 4PPthPP ECOOP workshop on Mobile Object Systems (MOS'98): Secure Internet Mobile Computations.

Holzle, U., Chambers, C., Ungar, D. (juin 1992). Debugging Optimized Code with Dynamic Deoptimization. Dans ACM SIGPLAN 92 Conference on Programming Language Design and Implementation (PLDI'92); San Francisco, CA, USA.

Huang, Y., Kintala, C. et Wang, Y-M. (1995). Software Tools and Libraries for Fault-Tolerance. IEEE Technical Commitee on Operating Systems and Application Environments, Volume 7, Numéro 4.

Hunter, M. J. et Knightbridge, P. (1988). An Overview of HARP Version 4.3, Esprit Project N0 1228 (Chameleon). Dans Technical Report.

Huq, R., Lacheray, H., Fulford, C., Wight, D.,et Apkarian, J. (2009). Qbot: an Educational Mobile Robot Controlled in Matlab Simulink Environment. Dans IEEE.

IEEE Foundation for Intelligent Physical Agents (2011). En ligne http://www.fipa.org/ (page consultée le 10 août 2013)

Illmann, T., Krueger, T., Kargl, F. et Weber, M. (décembre 2001). Transparent Migration of Mobile Agents Using the Java Debugger Architecture. Dans The Fifth IEEE International Conference on Mobile Agents (MA'2001), Atlanta, Géorgie, États-Unis.

Intel Corporation (1988). Hexadecimal Object File Format Specification (Revision A). En ligne http://microsym.com/editor/assets/intelhex.pdf (page consultée le 10 août 2013).

Islam, N., Mallah, G. A. et Shaikh, Z. A. (2010). FIPA and MASIF Standards: A Comparative Study and Strategies for Integration. Dans Proceeding NSEC '10 Proceedings of the 2010 National Software Engineering Conference, Article No. 7, ACM New York, NY, USA.

Ismail, L. et Hagimont, D. (1999). A Performance Evaluation of the Mobile Agent Paradigm. Dans Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 306-313.

Izatt, M., Chan, P., Brecht, T. (2000). Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Applications. Dans Concurrency: Practice and Experience, 12(8).

Johansen, D. (2004). Mobile agents: Right concept, wrong approach. Dans Mobile Data Management.

Johansen, D. (septembre 1998). Mobile Agent Applicability. Dans 2PPndPP International Workshop on Mobile Agents, MA'98, Stuttgart, Germany, p. 80-98, Lecture Notes in Computer Science, nPPoPP 477.

Joy, B., Steele, G., Gosling, J., et Bracha, G. (2000). Dans The Java Language Speci_cation. Addison-Wesley Pub Co, 2nd edition.

Jun, L., Xianlang, L., Hong, H. et Xu, Z. (2002). Application of mobile agent in wide area network. Dans IEEE.

Kawaguchi, N., Toyama, K., Inagaki, Y. (avril 2000). Magnet: Adhoc network system based on mobile agents. Dans Computer Communications. v.23, n.8, 2000, p.761-768, Elsevier.

Killijian, M.-O. (janvier 2000). Tolérance aux Fautes sur CORBA par Protocole à Métaobjets et Langages Réflexifs. Dans Thèse de Doctorat, Institut National Polytechnique de Toulouse, France.

Knightbridge, P. et Hunter, M. J. (1987). Harp 4, Esprit Project N0 1228 (Chameleon). Dans Technical Report.

Kotz, D. et Gray, R. S. (juillet 1999). Mobile agents and the future of the internet. Dans Operating Systems Review, 33(3):7–13.

Kulyukin, V., Sute, P. et Graw, N. D. (mars 2004a). Human robot interaction in a robotic guide for the visually impaired. Dans Proc. of the AAAI Spring Symposium on Interaction between Humans and Autonomous Systems over Extended Operation.

Kulyukin, V., Gharpure, C., Nicholson, J. et Pavithran S. (septembre 2004b). Rfid in robotassisted indoor navigation for the visually impaired. Dans Proceedings of the 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems.

Kulyukin, V., Gharpure, C., Sute, P., Graw, N. D. and Nicholson, J. (2004c). A robotic waynding system for the visually impaired. In Proceedings of the Sixteenth Innovative Applications of Articial Intelligence Conference (IAAI-04).

Kusek, M. et Jezic, G. (février 2005). Modeling agent mobility with UML sequence diagram. Dans Agent-Oriented Software Engineering TFG, AL3.

Labrosse, J. J., Benavides, J. P., Fernandez-Villasenor, J. (avril 2011). µC/OS-III: The Real-Time Kernel and the Freescale Kinetis ARM Cortex-M4. Dans Micrium Press, Weston, USA, 1020 pages.

Labrosse, J.J. (2009). µC/OS-III, The Real-Time Kernel. Dans Micrium Press, Weston, USA, 820 pages.

Labrosse J. J. (2002). Micro/OS-II The Real-Time and Implementation, second Edition. Dans CMP Books, San Francisco, CA.

Ladetto, Q., Merminod, B. (2002). Digital Magnetic Compass and Gyroscope Integration for Pedestrian Navigation. Dans 9th St-Petersburg International Conference on Integrated Navigation Systems.

Lange, D. B. et Oshima, M. (mars 1999). Seven Good Reasons for Mobile Agents. Dans Communications of the ACM, Vol. 42, No. 3.

Lange, D. B. et Oshima, M. (1998). Programming and Deploying Java Mobile Agents Aglets 1st edition. Dans Boston, MA: Addison-Wesley Longman Publishing Co., Inc.

Laprie, J. C. (1995). Guide de la sûreté de fonctionnement. Dans Cépatuès Édition, LIS: Laboratoire d'Ingénierie de la Sûreté de fonctionnement.

Lawall, J. L. et Muller, G. (juin 2000). Efficient Incremental Checkpointing of Java Programs. Dans International Conference on Dependable Systems and Networks (DSN'2000), New York, États-Unis.

Libsie, M. et Kosch, H. (2002). Content adaptation of multimedia delivery and indexing using MPEG-7. Dans Proceedings of the tenth ACM international conference on Multimedia (MM-02), pages 644–646, New York, ACM Press.

Lindholm, T., Yellin, F. (1999). The Java Virtual Machine Specification (2nd edition). Dans Addison Wesley.

Litzkow, M. J., Solomo, M. (janvier 1992). Supporting Checkpointing and Process Migration outside the UNIX Kernel. Dans USENIX Winter Conference; San Francisco, CA, USA.

Litzkow, M., Livny, M. (1990). Experience with the Condor distributed batch system. Dans IEEE Workshop on Experience Distributed Systems.

Litzkow, M., Livny, M., et Mutka, M. (1988). Condor – a hunter of idle workstations. Dans Proceedings of the 8th International Conference on Distributed Computing.

Liu, A. L, Hile, H., Kautz, H., Borriello, G., Brown, P.A, Harniss, M. et Johnson, K. (2006). Indoor wayfinding: developing a functional interface for individuals with cognitive impairments. Dans Proc. of the 8th international ACM SIGACCESS conference on Computers and accessibility, NY, NY, pages 95-102.

Lohmann, D., Scheler, F., Tartler, R., Spinczyk, O., et Schröder-Preikschat, W. (avril 2006). A Quantitative Analysis of Aspects in the eCos Kernel. Dans EuroSys Conference Leuven, Belgique.

Loukil, A., Hachicha, H. Hachicha et Ghedira, K. (mars 2006). A proposed approach to model and to implement mobile agents. Dans International Journal of Computer Science and Network Security (IJCSNS), 6(3B):125–129.

Loulou Aloulou, M. (2010). Approche Formelle pour la Spécification, la Vérification et le Déploiement des Politiques de Sécurité Dynamiques dans les Systèmes à base d'Agents Mobiles. Dans la thèse de doctorat en cotutelle de l'Université de Bordeaux 1 (France) et de l'Université de Sfax (Algérie).

Maes, P. (1987). Concepts and Experiments in Computational Reflection. Dans Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87), pages 147-155.

McCann, P. J. et Roman, G. C. (1998). Compositional programming abstractions for mobile computing. Dans IEEE Transactions on Software Engineering, 24(2):97–110.

McLean, J. (1994). Security Models. Dans Encyclopedia of Software Engineering (Ed. John Marciniak), Wiley and Sons, Inc.

Meloan, S. (juin 1999). The Java HotSpot Performance Engine: An In-Depth Look. Dans Sun Microsystems.

Menapace, J., Kingdon, J. et MacKenzie, D. (1993) The "stabs" Debug Format. Dans Technical report, Cygnus support.

Microchip Technology (2008). ENC28J60 Data Sheet, Stand-Alone Ethernet Controller with SPI Interface. En ligne http://ww1.microchip.com/downloads/en/devicedoc/39662c.pdf (page consultée le 10 août 2013).

Micrium (2013). µC/OS-II The Real-Time Kernel. En ligne http://micrium.com/rtos/ucosii/overview/ (page consultée le 13 août 2013).

Mifare Classic (2002). Smart Card Zone. En ligne http://www.smartcardzone.com/mifare4k.asp (page consultée le 13 août 2013).

Milojicic, D.S., Douglis, F., Paindaveine, Weeler, Y., R., et Zhou, S. (septembre 2000). Process migration. Dans ACM Compting Surveys, 32(3):241–299.

Milojicic, D., Breugst, M., Busse, I., Campbell, J., Covaci, S., Friedman, B., Kosaka, K., Lange, D, Ono, K., Oshima, M., Tham, C., Virdhagriswaran, S., White, J. (1998). MASIF—The OMG mobile agent system interoperability facility. Proceedings of the 2nd International Workshop of Mobile Agents (MA '98) (Lecture Notes in Computer Science, Vol. 1477), Rothermel K, Hohl F (eds.). Springer: Stuttgart; 50–67.

Milojicic, D. S. (avril 1993). Task Migration on top of the Mach Microkernel. Dans Proceeding of the 3rd USENIX Mach Symposium, Santa.

Milojicic, W., Zint, W. et Dangel, A. (1992). Task Migration on Top of the Mach Microkernel - Design and Implementation. Dans Technical report, University of Kaiserslautern, Allemagne.

Mobile Agents Working Group (2005). En ligne http://www.fipa.org/subgroups/MA-WG.html (page consultée le 13 août 2013).

Mobile-C (juin 2011). Mobile-C v2.1.4. En ligne http://www.mobilec.org/ (page consultée le 13 août 2013).

Muscutariu, F, et Gervais, M.P. (2001). On the modeling of mobile agent based systems. Dans Proceedings of the 3th International Workshop on Mobile Agents for Telecommunication Applications (MATA'01), volume 2164/2001 of LNCS, pages 219–233, Montreal, Canada.

Necula, G. C. et Lee, P. (1998). Safe, Untrusted Agent using Proof-Carrying Code. Dans Mobile Agent Security, ed. Par Vigna (Giovanni), p. 61-91, Lecture Notes in Computer Science, nPPoPP 1419.

Newhaven Display (novembre 2008). NHD-C12864EZ-FSW-FTW-P. En ligne http://www.newhavendisplay.com/specs/NHD-C12864EZ-FSW-FTW-P.pdf (page consultée le 10 août 2013).

Noack, M. (juillet 2003). Evaluation of Process Migration Algorithms. Dans Diploma Thesis Dresden, University of Technology, Sydney, Australie.

NXP (2012). UM10114 LPC21xx and LPC22xx User manual. En ligne http://www.nxp.com/documents/user_manual/UM10114.pdf (page consultée le 10 août 2013).

Object Management Group (janvier 2000). Mobile Agent Facility Specification. En ligne http://www.omg.org/cgi-bin/doc?formal/2000-01-02 (page consultée le 10 août 2013)

Olimex_a (2011). LPC-H2214. En ligne https://www.olimex.com/Products/ARM/NXP/ (page consultée le 10 janvier 2013).

Olimex_b (2011). LPC-H2294. En ligne https://www.olimex.com/Products/ARM/NXP/ (page consultée le 10 janvier 2013).

Oueichek, I. (octobre 1996). Conception et Réalisation d'un Noyau d'Administration pour un Système Réparti à Objets Persistants. Dans Thèse de Doctorat, Institut National Polytechnique de Grenoble, France.

Ousterhout, J. K, Cherenson, A. R., Douglis, F., Nelson, M. N. et Welch, B. B. (février 1988) The Sprite network operating system. IEEE Computer.

Papadakis, N., Doulamis, A., Litke, A., Doulamis, N., Skoutas, D. et Varvarigou, T. (2008). MI-MERCURY : A mobile agent architecture for ubiquitous retrieval and delivery of multimedia information. Multimedia Tools and Applications, 38(1):147–184.

Picco, G. P.(février 1998). Understanding, Evaluating, Formalizing, and Exploiting Code Mobility. Dans PhD thesis, Politecnico di Torino, Italy.

Pierre, S. (2011). Réseaux et systèmes informatiques mobiles, édition revue et augmentée - Fondements, architectures et applications. Dans Édition, Presses internationales Polytechnique, 632 pages.

Poggi, A., Rimassa, G., Turci, P., Odell, J., Mouratidis, H., et Manson, G. (2003). Modeling Deployment and Mobility Issues in Multiagent Systems Using AUML. Dans 4th International Workshop, Agent Oriented Software Engineering (AOSE 2003), pages 69-84, Melbourne, Australia.

Popovici, A., Alonso, G. et Gross, T. (2003). Just-In-Time Aspects: Efficient Dynamic Weaving for Java. Dans Proceeding AOSD '03 Proceedings of the 2nd international conference on Aspect-oriented software development, ACM New York, NY, USA, Pages 100 – 109.

Quang La, V. (2008). A study on Java Virtual Machine for Real-time embedded systems. Dans International Conference on Computer Science and Software Engineering.

Ran, L., Helal, S. et Moore, S. (2004). Drishti: An Integrated Indoor/Outdoor Blind Navigation System and Service. Dans Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04), pp. 23.

Rankl, W. (2007). Translated by Cox, K., Smart Card Applications Design Models for using and programming smart cards. Dans Chichester, England: John Wiley & Sons Ltd.

Rankl, W. et Effing, W. (2003). Translated by Cox, K., Smart Card Handbook, Third Edition. Dans John Wiley & Sons Ltd. Chichester, England.

Rashid, R. F. et Robertson, G. G. (1981). Accent: A communication oriented network operating system kernel. Dans Proceedings of the Ninth ACM Symposium on Operating System Principles.

Recursion Software (2011). Voyager Documentation. En ligne http://www.recursionsw.com/products/voyager/voyager-documentation.html (page consultée le 10 août 2013)

Renaudin, V., Yalak, O., Tomé, P. et Merminod, B.(juillet 2007). Indoor Navigation of Emergency Agents, European Journal of Navigation, Vol 5(3), pp. 36-45.

RFC 3411, (décembre 2002). An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks. En ligne https://www.ietf.org/rfc/rfc3411.txt (page consultée le 15 février 2014)

Richmond, M, A. (novembre 1996). Post-Copy Migration: A new process migration algorithm. Dans le mémoire de maîtrise (Master's thesis), University of Sydney, Australie.

Romito, B. (2012). Stockage décentralisé adaptatif: autonomie et mobilité des données dans les réseaux pair-à-pair. Dans thèse de doctorat Université de Caen Basse-Normandie, France.

Roth, V. (2004). Obstacles to the Adoption of Mobile Agents. Dans IEEE International Conference on Mobile Data Management, pages 296–297.

Roush, E. T. (1995). The freeze free algorithm for process migration. Dans la thèse de doctorat (Ph.D), University of Illinois at Urbana-Champaign.

Rozier, M. (avril 1992). Overview of the Chorus Distributed Operating System. Dans Proceedings of the USENIX Workshop on Microkernels and Other Kernel Architecture.

Ruiz-Garcia, J.-C. (décembre 2002). Contribution à la validation des systèmes réflexifs tolérants aux fautes: stratégie de test de protocoles à métaobjets. Dans thèse de doctorat de l'Institut national polytechnique de Toulouse (I.N.P.T.), France.

Sahai, A. et Morin , C. (mai 1998a). Mobile agents for enabling mobile user aware applications. Dans Proceedings of the 2nd International Conference on Autonomous Agents (Agents'98), pages 205–211, New York, ACM Press.

Sahai, A. et Morin, C. (1998). Enable a Mobile Networking Manager (MNM) through Mobile Agent. Dans Proceeding of the 2nd International Workshop on Mobile Agent, p. 249-260.

Sakamoto, T, Sekiguchi, T, Yonezawa, A. (septembre 2000). Bytecode Transformation for Portable Thread Migration in Java. 4th International Symposium on Mobile Agents 2000 (MA'2000), Zurich, Switzerland.

Satoh, I. (2002). Physical mobility and logical mobility in ubiquitous computing environments. Dans Lecture Notes in Computer Science, 2535:186–202.

Sekiguchi, T, Masuhara, H, Yonezawa, A. (avril 1999). A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation. Dans 3rd International Conference on Coordination Models and Languages, Amsterdam, The Netherlands.

Sewell, P., Wojciechowski, P. T. et Pierce, B. C. (1999). Location independent communication for mobile agents: A two-level architecture. Dans Lecture Notes in Computer Science Volume 1686, 1999, pages 1-31.

Shi, J., Wan, J., Yan, H. et Suo, H. (novembre 2011). A Survey of Cyber-Physical Systems. Dans In Proc. of the Int. Conf. on Wireless Communications and Signal Processing, Nanjing, China.

Siebert, F. (septembre 2007), Realtime Garbage Collection in the JamaicaVM 3.0. Dans Proceeding JTRES '07 Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems, pages 94-103, ACM New York, NY, USA.

Shub, C. M. (février 1990). Native Code Process-Originated Migration in a Heterogeneous Environment. Dans Proceedings of the 1990 ACM Annual Conference on Cooperation, p. 266-270, Washington, États-Unis.

Silberschatz, A., Galvin , P. B. et Gagne, G. Gagne (décembre 2012). Operating System Concepts, Ninth Edition. Dans John Wiley & Sons, Inc.

SoftIntegration (2011). Embedded Ch. En ligne http://www.softintegration.com/products/sdk/embedch/ (page consultée le 10 août 2013)

Smith, B. (1984). Refection and semantics in Lisp. Dans Proceedings of ACM Symposium on Principles of Programming Languages, pages 23-35.

Smith, B. (1982). Refection and semantics in a procedural language. Dans Technical Report 272, Laboratory for Computer Science, MIT.

Smith, J.M. (1995). A survey of process migration mechanisms. Dans Technical report, Columbia University, USA.

Smith, P. et Hutchinson, N. C. (mars 1997). Heterogeneous process Migration: the Tui System. Dans Rapport Technique, British Columbia University.

SpringCard (2009). K531-TTL product information sheet. En ligne http://www.springcard.com/download/pub/pfl81sp-ab.pdf (page consultée le 10 janvier 2013).

Steketee, C., Socko, P. et Kiepuszewski, B. (janvier 1996) Proceedings of the 19th ACSC Conference, Melbourne, Australia.

Steketee, C., Zhu, W., et Moseley, P. (1994). Implementation of Process Migration in Amoeba. In International Conference on Distributed Computing Systems.

Suezawa, T. (juin 2000). Persistent Execution State of a Java Virtual Machine. ACM Java Grande 2000 Conference; San Francisco, CA, USA.

Suganuma, T., Ogasawara,T., Takeuchi, M., Yasue, T., Kawahito, M., Ishizaki, K., Komatsu, H. et Nakatani, T. (2000) Overview of the IBM Java Just-in-Time Compiler. Dans IBM systems journal, VOL 39, NO 1, 2000

Sutandiyo, W., Chhetri, M.B., Krishnaswamy, S. et Loke, S. W. (avril 2004). Experiences with Software Engineering of Mobile Agent Applications. Dans Proceedings of the Australian Software Engineering Conference (ASWEC 2004), Melbourne, Australia, IEEE Press.

Sweet, M. R. (2011). mini-XML Programmers Manual Version 2.6. En ligne http://www.msweet.org/documentation/project3/Mini-XML.pdf (page consultée le 10 août 2013).

Taïani, F. (2004). La Réflexivité dans les architectures multiniveaux: application aux systèmes tolérant les fautes. Dans thèse de doctorat de l'Université Paul Sabatier de Toulouse, France.

Tanenbaum, A. (novembre 2008). Systèmes d'exploitation 3 édition. Dans Pearson Education.

Tanenbaum, A. S. (1989). Computer Networks. Dans édition, Prentice-Hall, p. 454-465.

Tanenbaum, A.S, van Staveren, H., Keizer, E.G., et Stevenson , J.W. (septembre 1983). A Practical Toolkit for Making Portable Compilers. Dans Communicationsof theACM, 26(9):654–660.

Tavernier, C. (2007). Les cartes à puce - Théorie et mise en œuvre - 2ème édition. Dans Édition dunod, 364 pages.

Theimer, M. M., Lantz, K. A. et Cheriton, D. R (décembre 1985). Preemptable Remote Execution Facilities for the V-System. Dans Proceedings of the 10th ACM Symposium on Operating System Principles.

Truyen, E, Robben, B, Vanhaute, B, Coninx, T, Joosen, W, Verbaeten, P. (septembre 2000) Portable Support for Transparent Thread Migration in Java. Dans 4th International Symposium on Mobile Agents 2000 (MA'2000); Zurich, Switzerland.

Urra, O., Ilarri, S., Mena, E. et Delot, T. (2009). Using hitchhiker mobile agents for environment monitoring. Dans Proceedings of the 7th International Conference on Practical Applications of Agents and Multi-Agent System, pages 557–566.

Young, A. et Yung, M. (1997). Sliding Encryption: A Cryptographic Tool for Mobile Agents. Dans Proceedings of Fast Software Encryption Workshop, p. 230-241, Lecture Notes in Computer Science, nPPoPPPP 1267.

Vigna, G. (janvier 2004). Mobile agents: Ten reasons for failure. Dans 5th IEEE International Conference on Mobile Data Management (MDM 2004), pages 298–299. IEEE Computer Society.

Wahbe, R., Lucco, S., Anderson, T. E. et Graham, S.L. (1993). Efficient Software-Based Fault Isolation. Dans Proceeding of the 14PPthPP ACM Symposium on Operating System Principles, pages 203-216.

Walder, T., Bernoulli, T. et Wießflecker, T. (mai 2009). An indoor positioning system for improved action force command and disaster management. Dans Proceedings of the 6th International ISCRAM Conference.

Watanabe, Y., Sato, S. et Ishida, Y. (2004). An Approach for Selfrepair in Distributed System Using Immunity-Based Diagnostic Mobile Agents. Dans KnowledgeBased Intelligent Information and Engineering Systems, volume 3214 de Lecture Notes dans Computer Science, pages 504–510.

Wilhelm, U. G, Buttyàn, L. et Staamann, S. (1999). Introducing Trusted Third Parties to the Mobile Agent Paradigm. Dans Secure Internet Programming: Security Issues for Mobile and Distributed Object, p. 471-491, New York, USA, Lecture Notes in Computer Science, nPPoPP 1603.

Wilhelm, U. G, Buttyàn, L. et Staamann, S. (mars 1998). On the problem of Trust in Mobile an Agent Systems. Dans Symposium on Network and Distributed System Security, p. 114-124, Internet Society.

Woodcock, J. et Davies, J. (1996). Using Z: Specification, Refinement and Proof. Dans International Thomson Computer Press, Upper Saddle River, NJ, USA.

Xiong, K. et Gao, Y. (septembre 2011). Designing and Porting Boot loader and uClinux with its Implementation based on LPC2478. Dans 7th International Conference on Wireless Communications, Networking and Mobile Computing (WiCOM).

Zayas, E. (avril 1987a). The Use of Copy-on-Reference in a Process Migration System. Dans la thèse de doctorat (Ph.D), Carnegie Mellon University.

Zayas, E. (avril 1987b). Attacking the process migration bottleneck. Dans Proceedings of the eleventh ACM Symposium on Operating systems principles, ACM Press, pages 13-24.

ZigBee Standards Organization (janvier 2008). ZigBee Specification. En ligne http://www.zigbee.org/ (page consultée le 10 août 2013).